ANHANG

Lösungen zu den Übungsaufgaben

Die Lösungen zu allen Programmierübungen sind online in den Jupyter Notebooks auf *https://homl.info/colab3* zu finden.

Kapitel 1: Die Machine-Learning-Umgebung

1. Beim Machine Learning geht es um das Konstruieren von Systemen, die aus Daten lernen können. Lernen bedeutet, sich bei einer Aufgabe anhand eines Qualitätsmaßes zu verbessern.
2. Machine Learning ist geeignet zum Lösen komplexer Aufgaben, bei denen es keine algorithmische Lösung gibt, zum Ersetzen langer Listen händisch erstellter Regeln, zum Erstellen von Systemen, die sich an wechselnde Bedingungen anpassen, und schließlich dazu, Menschen beim Lernen zu helfen (z.B. beim Data Mining).
3. Ein gelabelter Trainingsdatensatz ist ein Trainingsdatensatz, der die gewünschte Lösung (das Label) für jeden Datenpunkt enthält.
4. Die zwei verbreitetsten Aufgaben beim überwachten Lernen sind Regression und Klassifikation.
5. Verbreitete unüberwachte Lernaufgaben sind Clustering, Visualisierung, Dimensionsreduktion und das Erlernen von Assoziationsregeln.
6. Reinforcement Learning funktioniert wahrscheinlich am besten, wenn ein Roboter lernen soll, in unbekanntem Gelände zu laufen, da dies typischerweise die Art von Problem ist, die das Reinforcement Learning angeht. Die Aufgabe ließe sich auch als überwachte oder unüberwachte Aufgabe formulieren, diese Herangehensweise wäre aber weniger natürlich.
7. Wenn Sie nicht wissen, wie Sie die Gruppen definieren sollen, können Sie ein Clustering-Verfahren verwenden (unüberwachtes Lernen), um Ihre Kunden in

Cluster jeweils ähnlicher Kunden zu segmentieren. Kennen Sie die gewünschten Gruppen dagegen bereits, können Sie einem Klassifikationsalgorithmus viele Beispiele aus jeder Gruppe zeigen (überwachtes Lernen) und alle Kunden in diese Gruppen einordnen lassen.

8. Spamerkennung ist eine typische überwachte Lernaufgabe: Dem Algorithmus werden viele E-Mails und deren Labels (Spam oder Nicht-Spam) bereitgestellt.
9. Ein Onlinelernsystem kann im Gegensatz zu einem Batchlernsystem inkrementell lernen. Dadurch ist es in der Lage, sich sowohl an sich schnell ändernde Daten oder autonome Systeme anzupassen als auch sehr große Mengen an Trainingsdaten zu verarbeiten.
10. Out-of-Core-Algorithmen können riesige Datenmengen verarbeiten, die nicht in den Hauptspeicher des Computers passen. Ein Out-of-Core-Lernalgorithmus teilt die Daten in Mini-Batches ein und verwendet Techniken aus dem Online-Learning, um aus diesen Mini-Batches zu lernen.
11. Ein instanzbasiertes Lernsystem lernt die Trainingsdaten auswendig; anschließend wendet es ein Ähnlichkeitsmaß auf neue Datenpunkte an, um die dazu ähnlichsten erlernten Datenpunkte zu finden und diese zur Vorhersage zu verwenden.
12. Ein Modell besitzt einen oder mehrere Modellparameter, die festlegen, wie Vorhersagen für einen neuen Datenpunkt getroffen werden (z. B. die Steigung eines linearen Modells). Ein Lernalgorithmus versucht, optimale Werte für diese Parameter zu finden, sodass das Modell bei neuen Daten gut verallgemeinern kann. Ein Hyperparameter ist ein Parameter des Lernalgorithmus selbst und nicht des Modells (z. B. die Menge zu verwendender Regularisierung).
13. Modellbasierte Lernalgorithmen suchen nach einem optimalen Wert für die Modellparameter, sodass das Modell gut auf neue Datenpunkte verallgemeinert. Normalerweise trainiert man solche Systeme durch Minimieren einer Kostenfunktion. Diese misst, wie schlecht die Vorhersagen des Systems auf den Trainingsdaten sind, zudem wird im Fall von Regularisierung ein Strafterm für die Komplexität des Modells zugewiesen. Zum Treffen von Vorhersagen geben wir die Merkmale neuer Datenpunkte in die Vorhersagefunktion des Modells ein, wobei die vom Lernalgorithmus gefundenen Parameter verwendet werden.
14. Zu den Hauptschwierigkeiten beim Machine Learning gehören fehlende Daten, mangelhafte Datenqualität, nicht repräsentative Daten, nicht informative Merkmale, übermäßig einfache Modelle, die die Trainingsdaten underfitten, und übermäßig komplexe Modelle, die die Trainingsdaten overfitten.
15. Wenn ein Modell auf den Trainingsdaten herausragend abschneidet, aber schlecht auf neue Datenpunkte verallgemeinert, liegt vermutlich ein Overfitting der Trainingsdaten vor (oder wir hatten bei den Trainingsdaten eine

Menge Glück). Gegenmaßnahmen zum Overfitting sind das Beschaffen zusätzlicher Daten, das Vereinfachen des Modells (Auswählen eines einfacheren Algorithmus, Reduzieren der Parameteranzahl oder Regularisierung des Modells) oder das Verringern des Rauschens in den Trainingsdaten.

16. Ein Testdatensatz hilft dabei, den Verallgemeinerungsfehler eines Modells auf neuen Datenpunkten abzuschätzen, bevor ein Modell in einer Produktionsumgebung eingesetzt wird.
17. Ein Validierungsdatensatz wird zum Vergleichen von Modellen verwendet. Es ist damit möglich, das beste Modell auszuwählen und die Feineinstellung der Hyperparameter vorzunehmen.
18. Das Train-Dev-Set wird eingesetzt, wenn das Risiko besteht, dass es eine Diskrepanz zwischen den Trainingsdaten und den in den Validierungs- und Testdatensätzen verwendeten Daten gibt (die immer so nahe wie möglich an den Daten liegen sollten, die das Modell produktiv nutzen). Es ist Teil des Trainingsdatensatzes, der zurückgehalten wird (das Modell wird nicht damit trainiert). Stattdessen wird das Modell mit dem Rest des Trainingsdatensatzes trainiert und sowohl mit dem Train-Dev-Set wie auch mit dem Validierungsdatensatz evaluiert. Funktioniert das Modell mit dem Trainingsdatensatz gut, nicht aber mit dem Train-Dev-Set, ist es vermutlich für den Trainingsdatensatz overfittet. Funktioniert es gut mit dem Trainingsdatensatz und dem Train-Dev-Set, aber nicht mit dem Validierungsdatensatz, gibt es vermutlich einen signifikanten Unterschied zwischen Trainingsdaten einerseits und Validierungs- und Testdaten andererseits, und Sie sollten versuchen, die Trainingsdaten zu verbessern, damit diese mehr wie die Validierungs- und Testdaten aussehen.
19. Wenn Sie Hyperparameter mit den Testdaten einstellen, riskieren Sie ein Overfitting des Testdatensatzes. Der gemessene Verallgemeinerungsfehler ist dann zu niedrig angesetzt (Sie könnten in diesem Fall also ein Modell einsetzen, das schlechter funktioniert als erwartet).

Kapitel 2: Ein Machine-Learning-Projekt von A bis Z

Die Lösungen finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 3: Klassifikation

Die Lösungen finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 4: Trainieren von Modellen

1. Haben Sie einen Trainingsdatensatz mit Millionen Merkmalen, können Sie das stochastische Gradientenverfahren oder das Mini-Batch-Gradientenverfahren verwenden. Wenn die Trainingsdaten in den Speicher passen, funktioniert eventuell auch das Batch-Gradientenverfahren. Die Normalengleichung und auch der SVD-Ansatz funktionieren jedoch nicht, weil die Komplexität der Berechnung schnell (mehr als quadratisch) mit der Anzahl Merkmale ansteigt.
2. Wenn die Merkmale in Ihrem Trainingsdatensatz sehr unterschiedlich skaliert sind, hat die Kostenfunktion die Gestalt einer länglichen Schüssel. Deshalb benötigen die Algorithmen für das Gradientenverfahren lange zum Konvergieren. Um dieses Problem zu beheben, sollten Sie die Daten skalieren, bevor Sie das Modell trainieren. Die Normalengleichung und der SVD-Ansatz funktionieren auch ohne Skalierung. Darüber hinaus können regularisierte Modelle mit nicht skalierten Merkmalen bei einer suboptimalen Lösung konvergieren: Weil die Regularisierung große Gewichte abstraft, werden Merkmale mit geringen Beträgen im Vergleich zu Merkmalen mit großen Beträgen tendenziell ignoriert.
3. Das Gradientenverfahren kann beim Trainieren eines logistischen Regressionsmodells nicht in einem lokalen Minimum stecken bleiben, weil die Kostenfunktion konvex ist. Das bedeutet: Wenn Sie zwei beliebige Punkte der Kurve über eine gerade Linie verbinden, schneidet diese niemals die Kurve.
4. Ist das Optimierungsproblem konvex (wie bei der linearen oder logistischen Regression) und die Lernrate nicht zu hoch, finden sämtliche algorithmischen Varianten des Gradientenverfahrens das globale Optimum und führen zu sehr ähnlichen Modellen. Allerdings konvergieren das stochastische und das Mini-Batch-Gradientenverfahren nicht wirklich (es sei denn, Sie reduzieren die Lernrate), sondern springen um das globale Optimum herum. Das bedeutet, dass diese Algorithmen geringfügig unterschiedliche Modelle hervorbringen, selbst wenn Sie sie lange laufen lassen.
5. Sollte der Validierungsfehler nach jeder Epoche immer wieder steigen, ist die Lernrate möglicherweise zu hoch, und der Algorithmus divergiert. Wenn auch der Trainingsfehler steigt, ist dies mit Sicherheit die Ursache, und Sie sollten die Lernrate senken. Falls der Trainingsfehler aber nicht steigt, overfittet Ihr Modell die Trainingsdaten, und Sie sollten das Trainieren abbrechen.
6. Wegen des Zufallselements gibt es weder beim stochastischen noch beim Mini-Batch-Gradientenverfahren eine Garantie für Fortschritte bei jeder Iteration. Wenn Sie also das Trainieren abbrechen, sobald der Validierungsfehler steigt, kann es passieren, dass Sie vor Erreichen des Optimums abbrechen. Es ist günstiger, das Modell in regelmäßigen Abständen abzuspeichern und das beste gespeicherte Modell aufzugreifen, falls es sich über eine längere Zeit nicht verbessert (es also vermutlich den eigenen Rekord nicht knacken wird).

7. Die Trainingsiterationen sind beim stochastischen Gradientenverfahren am schnellsten, da dieses nur genau einen Trainingsdatenpunkt berücksichtigt. Es wird also normalerweise die Umgebung des globalen Optimums als Erstes erreichen (oder das Mini-Batch-Gradientenverfahren mit sehr kleinen Mini-Batches). Allerdings wird nur das Batch-Gradientenverfahren mit genug Trainingszeit auch konvergieren. Wie erwähnt, springen das stochastische und das Mini-Batch-Gradientenverfahren um das Optimum herum, es sei denn, Sie senken die Lernrate allmählich.
8. Wenn der Validierungsfehler deutlich höher als der Trainingsfehler ist, liegt es daran, dass Ihr Modell die Trainingsdaten overfittet. Dies lässt sich beheben, indem Sie den Grad des Polynoms senken: Ein Modell mit weniger Freiheitsgraden neigt weniger zu Overfitting. Sie können auch versuchen, das Modell zu regularisieren – beispielsweise über einen ℓ_2-Strafterm (Ridge) oder einen ℓ_1-Strafterm (Lasso), der zur Kostenfunktion addiert wird. Damit reduzieren Sie ebenfalls die Freiheitsgrade des Modells. Schließlich können Sie auch die Größe des Trainingsdatensatzes erhöhen.
9. Wenn der Trainingsfehler und der Validierungsfehler fast gleich und recht hoch sind, liegt vermutlich ein Underfitting der Trainingsdaten vor. Es gibt also ein hohes Bias. Sie sollten daher den Hyperparameter zur Regularisierung α senken.
10. Schauen wir einmal:
    - Ein Modell mit etwas Regularisierung arbeitet in der Regel besser als ein Modell ohne Regularisierung. Daher sollten Sie grundsätzlich die Ridge-Regression der einfachen linearen Regression vorziehen.
    - Die Lasso-Regression verwendet einen ℓ_1-Strafterm, wodurch Gewichte auf exakt null heruntergedrückt werden. Dadurch erhalten Sie spärliche Modelle, bei denen alle Gewichte außer den wichtigsten null sind. Auf diese Weise können Sie eine automatische Merkmalsauswahl durchführen, wenn Sie ohnehin schon den Verdacht hegen, dass nur einige Merkmale wichtig sind. Sind Sie sich nicht sicher, sollten Sie der Ridge-Regression den Vorzug geben.
    - Elastic Net ist grundsätzlich gegenüber der Lasso-Regression vorzuziehen, da sich Lasso in einigen Fällen sprunghaft verhält (wenn mehrere Merkmale stark miteinander korrelieren oder es mehr Merkmale als Trainingsdatenpunkte gibt). Allerdings gilt es, einen zusätzlichen Hyperparameter einzustellen. Wenn Sie Lasso ohne das sprunghafte Verhalten verwenden möchten, können Sie einfach Elastic Net mit einer `l1_ratio` um 1 verwenden.
11. Möchten Sie Bilder als außen/innen und Tag/Nacht klassifizieren, schließen sich die Kategorien nicht gegenseitig aus (d. h., alle vier Kombinationen sind möglich). Sie sollten daher zwei Klassifikatoren mit logistischer Regression trainieren.

Die Lösungen finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 5: Support Vector Machines

1. Der Grundgedanke bei Support Vector Machines ist, die breitestmögliche »Straße« zwischen den Kategorien zu fitten. Anders ausgedrückt, soll zwischen der Entscheidungsgrenze zwischen den beiden Kategorien und den Trainingsdatenpunkten eine möglichst große Lücke sein. Bei der Soft-Margin-Klassifikation sucht die SVM nach einem Kompromiss zwischen einer perfekten Trennung zwischen den Kategorien und der breitestmöglichen Straße (d.h., einige Datenpunkte dürfen auf der Straße liegen). Eine weiteres wichtiges Konzept ist die Verwendung von Kernels beim Trainieren nichtlinearer Datensätze. SVMs lassen sich auch anpassen, um lineare und nichtlineare Regression sowie eine Ausreißererkennung durchzuführen.
2. Nach dem Trainieren einer SVM ist jeder Datenpunkt an der »Straße« (siehe vorherige Antwort) ein *Stützvektor*, einschließlich des Straßenrands. Die Entscheidungsgrenze ist vollständig durch die Stützvektoren festgelegt. Jeder Datenpunkt, der *kein* Stützvektor ist (d.h. abseits der Straße liegt), hat darauf keinen Einfluss; Sie könnten diese entfernen, weitere Datenpunkte hinzufügen oder sie verschieben. Solange sie weg von der Straße bleiben, beeinflussen sie die Entscheidungsgrenze nicht. Zum Berechnen einer Vorhersage mit einer Kernelized SVM sind nur die Stützvektoren nötig, nicht der gesamte Datensatz.
3. SVMs versuchen, die breitestmögliche »Straße« zwischen den Kategorien einzufügen (siehe erste Antwort). Wenn also die Trainingsdaten nicht skaliert sind, neigt die SVM dazu, kleine Merkmale zu ignorieren (siehe Abbildung 5-2).
4. Sie können die Methode `decision_function()` nutzen, um Konfidenzwerte zu erhalten. Diese Werte repräsentieren den Abstand zwischen einem Testdatenpunkt und der Entscheidungsgrenze. Allerdings lassen sie sich nicht direkt in eine Schätzung der Wahrscheinlichkeit einer Kategorie umrechnen. Wenn Sie beim Erstellen einer SVM in Scikit-Learn `probability=True` einstellen, wird nach dem Training eine fünffache Kreuzvalidierung genutzt, um Out-of-Sample-Scores für die Trainingsdaten zu erzeugen, zudem wird ein logistisches Regressionsmodell trainiert, um diese Scores auf die geschätzten Wahrscheinlichkeiten abzubilden. Damit stehen dann die Methoden `predict_proba()` und `predict_log_proba()` zur Verfügung.
5. Alle drei Klassen lassen sich für eine lineare Large-Margin-Klassifikation nutzen. Die Klasse `SVC` unterstützt zudem den Kerneltrick, wodurch sie auch für nichtlineare Aufgaben geeignet ist. Aber das gibt es nicht umsonst: Die Klasse `SVC` skaliert nicht gut für Datensätze mit vielen Instanzen. Mit vielen Merkmalen hingegen lässt sie sich gut einsetzen. Die Klasse `LinearSVC` implementiert einen optimierten Algorithmus für lineare SVMs, während `SGDClassifier` das stochastische Gradientenverfahren einsetzt. Abhängig vom Datensatz kann `LinearSVC` ein wenig schneller als `SGDClassifier` sein (muss es aber nicht), während `SGDClassifier` flexibler ist und zudem inkrementelles Lernen unterstützt.

6. Wenn ein mit einem RBF-Kernel trainierter SVM-Klassifikator die Trainingsdaten underfittet, gibt es möglicherweise zu viel Regularisierung. Um diese zu senken, müssen Sie `gamma` oder `C` erhöhen (oder beide).
7. Ein Regressions-SVM-Modell versucht, so viele Instanzen wie möglich in einem kleinen Bereich um seine Vorhersagen herum anzupassen. Fügen Sie in diesem Bereich weitere Instanzen hinzu, wird das Modell dadurch nicht beeinflusst – es ist *ε-insensitiv*.
8. Der Kerneltrick ist eine mathematische Technik, die es ermöglicht, ein nichtlineares SVM-Modell zu trainieren. Das sich so ergebende Modell entspricht dem Abbilden der Eingaben mithilfe einer nichtlinearen Transformation auf einen anderen Raum und dann dem Trainieren eines linearen SVM auf den sich ergebenden hochdimensionalen Eingabewerten. Der Kerneltrick liefert das gleiche Ergebnis, nur ohne die Notwendigkeit, die Eingaben überhaupt transformieren zu müssen.

Die Lösungen zu den Übungsaufgaben 9, 10 und 11 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 6: Entscheidungsbäume

1. Die Tiefe eines ausbalancierten Binärbaums mit m Blättern ist $\log_2(m) = \log(m) / \log(2)$. Ein binärer Entscheidungsbaum (der wie alle Bäume in Scikit-Learn nur binäre Entscheidungen trifft) ist nach dem Trainieren mehr oder weniger ausbalanciert und enthält ein Blatt pro Trainingsdatenpunkt, falls er ohne Einschränkungen trainiert wird. Wenn also der Trainingsdatensatz eine Million Datenpunkte enthält, hat der Binärbaum eine Tiefe von $\log_2(10^6) \approx 20$ (in der Praxis ein wenig mehr, da der Baum nicht perfekt ausbalanciert sein wird).
2. Die Gini-Unreinheit eines Knotens ist im Allgemeinen geringer als die des Elternknotens. Dies liegt daran, dass die Kostenfunktion des CART-Trainingsalgorithmus jeden Knoten so aufteilt, dass die gewichtete Summe der Gini-Unreinheiten der Kinder minimal wird. Es ist aber möglich, dass ein Knoten eine höhere Gini-Unreinheit als der Elternknoten erhält, solange dies durch eine geringere Gini-Unreinheit seines Geschwisterknotens ausgeglichen wird. Betrachten Sie beispielsweise einen Knoten mit vier Datenpunkten aus Kategorie A und einem aus Kategorie B. Dessen Gini-Unreinheit beträgt $1 – (1/5)^2 – (4/5)^2 = 0{,}32$. Nehmen Sie an, dass der Datensatz eindimensional ist und die Datenpunkte in folgender Reihenfolge liegen: A, B, A, A, A. Es lässt sich nachweisen, dass der Algorithmus diesen Knoten nach dem zweiten Datenpunkt aufteilt und somit ein Kind mit den Datenpunkten A, B und eines mit den Datenpunkten A, A, A entsteht. Die Gini-Unreinheit des ersten Kinds beträgt $1 – (1/2)^2 – (1/2)^2 = 0{,}5$, was höher als die des Elternknotens ist. Dies wird dadurch kompensiert, dass der zweite Knoten rein ist, die gesamte gewichtete

Gini-Unreinheit beträgt damit 2/5 × 0,5 + 3/5 × 0 = 0,2, was geringer als die Gini-Unreinheit des Elternknotens ist.

3. Wenn ein Entscheidungsbaum die Trainingsdaten overfittet, sollten Sie eventuell `max_depth` verringern, da diese Einschränkung das Modell regularisiert.
4. Entscheidungsbäume scheren sich nicht darum, ob die Trainingsdaten skaliert oder zentriert sind, dies ist eine ihrer angenehmen Eigenschaften. Wenn also ein Entscheidungsbaum die Trainingsdaten underfittet, ist das Skalieren der Eingabemerkmale reine Zeitverschwendung.
5. Die Komplexität der Berechnung beim Trainieren eines Entscheidungsbaums beträgt $O(n \times m \log_2(m))$. Wenn Sie also die Größe des Trainingsdatensatzes mit 10 multiplizieren, verlängert sich die Zeit zum Trainieren um den Faktor $K = (n \times 10m \times \log_2(10m)) / (n \times m \times \log_2(m)) = 10 \times \log_2(10m) / \log_2(m)$. Bei $m = 10^6$ beträgt $K \approx 11{,}7$, Sie können also mit einer Trainingsdauer von etwa 11,7 Stunden rechnen.
6. Verdoppelt sich die Anzahl der Merkmale, wird sich die Trainingszeit auch ungefähr verdoppeln.

Die Lösungen zu den Übungsaufgaben 7 und 8 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 7: Ensemble Learning und Random Forests

1. Wenn Sie fünf unterschiedliche Modelle trainiert haben und alle eine Relevanz von 95% erzielen, können Sie diese zu einem Ensemble kombinieren, was häufig zu noch besseren Ergebnissen führt. Unterscheiden sich die Modelle sehr stark, funktioniert es noch besser (z.B. ein SVM-Klassifikator, ein Entscheidungsbaum, ein Klassifikator mit logistischer Regression und so weiter). Durch Trainieren auf unterschiedlichen Trainingsdaten lässt sich eine weitere Verbesserung erzielen (darum geht es beim Bagging und Pasting von Ensembles), aber es wird auch ohne effektiv sein, solange die Modelle sehr unterschiedlich sind.
2. Ein Klassifikator mit Hard Voting zählt einfach nur die Stimmen jedes Klassifikators im Ensemble und wählt die Kategorie aus, die die meisten Stimmen erhält. Ein Klassifikator mit Soft Voting berechnet den Durchschnitt der geschätzten Wahrscheinlichkeiten für jede Kategorie und wählt die Kategorie mit der höchsten Wahrscheinlichkeit aus. Damit erhalten Stimmen mit hoher Konfidenz mehr Gewicht, was oft besser funktioniert. Dies gelingt aber nur, wenn jeder Klassifikator zum Abschätzen von Wahrscheinlichkeiten in der Lage ist (z.B. bei SVM-Klassifikatoren in Scikit-Learn müssen Sie `probability=True` setzen).
3. Es ist möglich, das Trainieren eines Ensembles mit Bagging durch Verteilen auf mehrere Server zu beschleunigen, da jeder Prädiktor im Ensemble unabhängig von den anderen ist. Aus dem gleichen Grund gilt dies auch für

Ensembles mit Pasting und Random Forests. Dagegen baut jeder Prädiktor in einem Boosting-Ensemble auf dem vorherigen Prädiktor auf, daher ist das Trainieren notwendigerweise sequenziell, und ein Verteilen auf mehrere Server nutzt nichts. Bei Stacking-Ensembles sind alle Prädiktoren einer Schicht unabhängig voneinander und lassen sich daher parallel auf mehreren Servern trainieren. Allerdings lassen sich die Prädiktoren einer Schicht erst trainieren, nachdem die vorherige Schicht vollständig trainiert wurde.

4. Bei der Out-of-Bag-Evaluation wird jeder Prädiktor in einem Bagging-Ensemble mit Datenpunkten ausgewertet, auf denen er nicht trainiert wurde (diese wurden zurückgehalten). Damit ist eine recht unbeeinflusste Evaluation des Ensembles ohne einen zusätzlichen Validierungsdatensatz möglich. Dadurch stehen Ihnen also mehr Trainingsdaten zur Verfügung, und Ihr Ensemble verbessert sich leicht.
5. Beim Erzeugen eines Baums in einem Random Forest wird beim Aufteilen eines Knotens nur eine zufällig ausgewählte Untermenge der Merkmale berücksichtigt. Dies gilt auch bei Extra-Trees, diese gehen aber noch einen Schritt weiter: Anstatt wie gewöhnliche Entscheidungsbäume nach dem bestmöglichen Schwellenwert zu suchen, verwenden sie für jedes Merkmal zufällige Schwellenwerte. Dieses zusätzliche Zufallselement wirkt wie eine Art Regularisierung: Wenn ein Random Forest die Trainingsdaten overfittet, könnten Extra-Trees besser abschneiden. Da außerdem Extra-Trees nicht nach dem bestmöglichen Schwellenwert suchen, lassen sie sich viel schneller trainieren als Random Forests. Allerdings sind sie beim Treffen von Vorhersagen weder schneller noch langsamer als Random Forests.
6. Wenn Ihr AdaBoost-Ensemble die Trainingsdaten underfittet, können Sie die Anzahl der Estimatoren steigern und die Regularisierung des zugrunde liegenden Estimators über dessen Hyperparameter verringern. Sie können auch versuchen, die Lernrate ein wenig zu erhöhen.
7. Wenn Ihr Gradient-Boosting-Ensemble die Trainingsdaten overfittet, sollten Sie die Lernrate senken. Sie können auch Early Stopping verwenden, um die richtige Anzahl Prädiktoren zu finden (vermutlich haben Sie zu viele davon).

Die Lösungen zu den Übungsaufgaben 8 und 9 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 8: Dimensionsreduktion

1. Gründe zum Einsatz und Nachteile:
   - Die Hauptgründe zum Einsatz von Dimensionsreduktion sind:
     - Die nachfolgenden Trainingsalgorithmen zu beschleunigen (in manchen Fällen sogar Rauschen und redundante Merkmale zu entfernen, wodurch das Trainingsverfahren eine höhere Leistung erbringt).

    - Die Daten zu visualisieren und Einblick in ihre wichtigsten Eigenschaften zu erhalten.
    - Platz zu sparen (Kompression).
  - Die wichtigsten Nachteile sind:
    - Es geht Information verloren, wodurch die Leistung nachfolgender Trainingsverfahren möglicherweise sinkt.
    - Sie kann rechenintensiv sein.
    - Sie erhöht die Komplexität Ihrer maschinellen Lernpipelines.
    - Transformierte Merkmale sind oft schwer zu interpretieren.
2. Der »Fluch der Dimensionalität« beschreibt den Umstand, dass in höher dimensionalen Räumen viele Schwierigkeiten auftreten, die es bei weniger Dimensionen nicht gibt. Beim Machine Learning ist eine häufige Erscheinungsform, dass zufällig ausgewählte höher dimensionale Vektoren grundsätzlich weit voneinander entfernt sind, was das Risiko für Overfitting erhöht und das Erkennen von Mustern in den Daten erschwert, wenn nicht sehr viele Trainingsdaten vorhanden sind.
3. Sobald die Dimensionalität eines Datensatzes mit einem der besprochenen Algorithmen verringert wurde, ist es so gut wie immer unmöglich, den Vorgang vollständig umzukehren, weil im Laufe der Dimensionsreduktion Information verloren geht. Während es bei einigen Algorithmen (wie der PCA) eine einfache Prozedur zur reversen Transformation gibt, mit der sich der Datensatz recht nah am Original rekonstruieren lässt, ist dies bei anderen Algorithmen (wie T-SNE) nicht möglich.
4. Mit der Hauptkomponentenzerlegung (PCA) lässt sich die Anzahl der Dimensionen der meisten Datensätze erheblich reduzieren, selbst wenn sie stark nichtlinear sind, weil sie zumindest die bedeutungslosen Dimensionen verwerfen kann. Wenn es aber keine bedeutungslosen Dimensionen gibt – wie beispielsweise beim Swiss-Roll-Datensatz –, geht bei der Dimensionsreduktion mittels PCA zu viel Information verloren. Sie möchten die Swiss Roll aufrollen, nicht platt quetschen.
5. Dies ist eine Fangfrage: Es kommt auf den Datensatz an. Betrachten wir zwei Extrembeispiele. Angenommen, der Datensatz bestünde aus beinahe perfekt aufgereihten Datenpunkten. In diesem Fall kann die Hauptkomponentenzerlegung den Datensatz auf nur eine Dimension reduzieren und trotzdem 95 % der Varianz erhalten. Wenn dagegen der Datensatz aus perfekt zufällig angeordneten Punkten besteht, die überall in den 1.000 Dimensionen verstreut sind, sind etwa 950 Dimensionen nötig, um 95 % der Varianz zu erhalten. Die Antwort ist also, dass es auf den Datensatz ankommt und dass es eine beliebige Zahl zwischen 1 und 950 sein kann. Eine Möglichkeit, die intrinsische Dimensionalität des Datensatzes zu verdeutlichen, ist, die abgedeckte Varianz über der Anzahl der Dimensionen zu plotten.

6. Gewöhnliche Hauptkomponentenzerlegung ist Standard, sie funktioniert aber nur, wenn der Datensatz in den Speicher passt. Die inkrementelle PCA ist bei großen Datensätzen, die nicht in den Speicher passen, hilfreich, sie ist aber langsamer als die gewöhnliche PCA. Wenn der Datensatz in den Speicher passt, sollten Sie also die gewöhnliche Hauptkomponentenzerlegung vorziehen. Die inkrementelle PCA ist auch bei Onlineaufgaben nützlich, bei denen eine PCA bei neu eintreffenden Daten jedes Mal im Vorübergehen durchgeführt werden soll. Die randomisierte PCA ist nützlich, wenn Sie die Anzahl der Dimensionen erheblich reduzieren möchten und der Datensatz in den Speicher passt; in diesem Fall ist sie deutlich schneller als die gewöhnliche PCA. Schließlich ist die Zufallsprojektion bei sehr hochdimensionalen Datensätzen hilfreich.
7. Intuitiv arbeitet ein Algorithmus zur Dimensionsreduktion dann gut, wenn er eine Menge Dimensionen aus dem Datensatz entfernt, ohne zu viel Information zu vergeuden. Dies lässt sich beispielsweise durch Anwenden der reversen Transformation und Bestimmen des Rekonstruktionsfehlers messen. Allerdings unterstützen nicht alle Verfahren zur Dimensionsreduktion die reverse Transformation. Wenn Sie die Dimensionsreduktion als Vorverarbeitungsschritt vor einem anderen Machine-Learning-Verfahren verwenden (z.B. einem Random-Forest-Klassifikator), können Sie auch einfach die Leistung des nachgeschalteten Verfahrens bestimmen; sofern bei der Dimensionsreduktion nicht zu viel Information verloren geht, sollte der Algorithmus genauso gut abschneiden wie auf dem ursprünglichen Datensatz.
8. Es kann durchaus sinnvoll sein, zwei unterschiedliche Algorithmen zur Dimensionsreduktion in Reihe zu schalten. Ein verbreitetes Beispiel ist eine Hauptkomponentenzerlegung oder eine Zufallsprojektion, um schnell eine große Anzahl unnützer Dimensionen loszuwerden und anschließend einen deutlich langsameren Algorithmus zur Dimensionsreduktion wie LLE zu verwenden. Dieser zweistufige Prozess führt vermutlich ungefähr zur gleichen Qualität wie LLE für sich allein, benötigt aber nur einen Bruchteil der Zeit.

Die Lösungen zu den Übungsaufgaben 9 und 10 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 9: Techniken des unüberwachten Lernens

1. Im Machine Learning ist Clustering die unüberwachte Aufgabe, ähnliche Instanzen zu gruppieren. Der Begriff der Ähnlichkeit hängt von der aktuellen Aufgabe ab: In manchen Fällen werden zwei nahe beieinanderliegende Instanzen als ähnlich angesehen, während in anderen ähnliche Instanzen weit voneinander entfernt sein können, solange sie zur gleichen dicht gepackten Gruppe gehören. Zu den verbreiteten Clustering-Algorithmen gehören K-Means, DBSCAN,

agglomeratives Clustern, BIRCH, Mean-Shift, Affinity Propagation und spektrales Clustern.

2. Zu den Hauptanwendungsgebieten von Clustering-Algorithmen gehören Datenanalyse, Kundensegmentierung, Empfehlungssysteme, Suchmaschinen, Bildsegmentierung, teilüberwachtes Lernen, Dimensionsreduktion, Anomalieerkennung und Novelty Detection.
3. Die Ellenbogenregel ist eine einfache Technik, um beim Einsatz von K-Means die Anzahl an Clustern festzulegen: Tragen Sie einfach die Trägheit (den mittleren quadratischen Abstand jeder Instanz zu ihrem nächstgelegenen Schwerpunkt) gegen die Anzahl der Cluster auf und finden Sie den Punkt der Kurve, bei dem die Trägheit nicht mehr schnell fällt (den »Ellenbogen«). Das liegt im Allgemeinen nahe an der optimalen Anzahl an Clustern. Eine andere Möglichkeit ist, den Silhouettenkoeffizienten als Funktion der Anzahl von Clustern auszugeben. Es gibt oft ein Maximum, und die optimale Anzahl an Clustern liegt meist dort in der Nähe. Der Silhouettenkoeffizient ist der Mittelwert der Silhouetten aller Instanzen. Dieser Wert liegt zwischen +1 für Instanzen, die gut in ihren Clustern und fern anderer Cluster liegen, und –1 für Instanzen, die sich sehr nahe an anderen Clustern befinden. Sie können auch die Silhouettendiagramme ausgeben und eine umfassendere Analyse durchführen.
4. Es ist teuer und zeitaufwendig, einen Datensatz mit Labels auszustatten. Daher hat man häufig viele ungelabelte Instanzen und nur wenige mit Labels. Label Propagation ist eine Technik, bei der manche (oder alle) Labels von den gelabelten auf ähnliche ungelabelte Instanzen übertragen werden. Das kann die Menge an gelabelten Instanzen deutlich vergrößern und es damit einem überwachten Algorithmus ermöglichen, eine bessere Performance zu liefern (das ist eine Form des teilüberwachten Lernens). Ein Ansatz ist die Verwendung eines Clustering-Algorithmus wie K-Means für alle Instanzen. Danach wird für jedes Cluster das verbreitetste Label oder das der repräsentativsten Instanz genutzt (zum Beispiel der Instanz, die am nächsten am Schwerpunkt liegt) und auf die ungelabelten Instanzen des gleichen Clusters übertragen.
5. K-Means und BIRCH skalieren sehr gut bei großen Datensätzen. DBSCAN und Mean-Shift suchen nach Regionen mit hoher Dichte.
6. Aktives Lernen ist immer dann nützlich, wenn Sie viele ungelabelte Instanzen haben, das Labeln aber teuer ist. In diesem (recht häufig vorkommenden) Fall ist es oft besser, nicht zufällig ausgewählte Instanzen mit Labels zu versehen, sondern ein aktives Lernen durchzuführen, bei dem menschliche Experten mit dem Lernalgorithmus interagieren und Labels für spezifische Instanzen liefern, wenn der Algorithmus diese anfordert. Häufig wird dazu Uncertainty Sampling eingesetzt (siehe die Beschreibung unter »Aktives Lernen« in Kapitel 9).
7. Die Begriffe *Anomalieerkennung* und *Novelty Detection* werden oft für das Gleiche verwendet, auch wenn sie nicht ganz genau das Gleiche beschreiben.

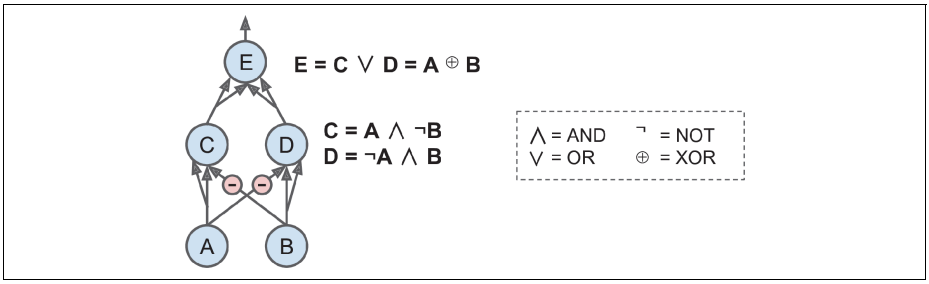
Bei der Anomalieerkennung wird der Algorithmus mit einem Datensatz trainiert, der Ausreißer enthalten kann. Das Ziel ist normalerweise, diese Ausreißer (im Trainingsdatensatz) und solche in neuen Instanzen zu identifizieren. Bei der Novelty Detection wird der Algorithmus mit einem Datensatz trainiert, der als »sauber« angenommen wird. Ziel ist hier, Ausreißer (Novelties) nur in neuen Instanzen zu finden. Manche Algorithmen funktionieren am besten bei der Anomalieerkennung (zum Beispiel Isolation Forest), andere sind besser zur Novelty Detection geeignet (zum Beispiel One-Class-SVMs).

8. Ein gaußsches Mischverteilungsmodell (GMM) ist ein Wahrscheinlichkeitsmodell, das annimmt, dass die Instanzen aus einer Mischung mehrerer Gaußverteilungen erzeugt wurden, deren Parameter unbekannt sind. Oder anders gesagt: Es wird davon ausgegangen, dass die Daten in einer endlichen Zahl von Clustern gruppiert sind, von denen jedes eine ellipsoide Form hat (das aber je nach Cluster eine andere Größe, Ausrichtung und Dichte haben kann), und wir wissen nicht, zu welchem Cluster jede Instanz gehört. Dieses Modell ist für die Dichteabschätzung, das Clustering und die Anomalieerkennung nützlich.
9. Sie können die richtige Zahl an Clustern mit einem gaußschen Mischverteilungsmodell unter anderem herausfinden, indem Sie das bayessche Informationskriterium (BIC) oder das Akaike-Informationskriterium (AIC) als Funktion der Anzahl von Clustern auftragen und dann die Zahl wählen, die das BIC oder AIC minimiert. Eine andere Technik ist der Einsatz eines bayesschen gaußschen Mischverteilungsmodells, das automatisch die Zahl der Cluster bestimmt.

Die Lösungen zu den Übungen 10 bis 13 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 10: Einführung in künstliche neuronale Netze mit Keras

1. Suchen Sie den TensorFlow Playground (*https://playground.tensorflow.org/*) auf und spielen Sie mit ihm herum (wie in der Übung beschrieben).
2. Hier ist ein aus den ursprünglichen künstlichen Neuronen aufgebautes neuronales Netz, das $A \oplus B$ berechnet (wobei $\oplus$ für das exklusive OR steht). Es macht sich den Umstand zunutze, dass $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. Es gibt weitere Lösungsmöglichkeiten – beispielsweise mithilfe von $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$ oder $A \oplus B = (A \vee B) \wedge (\neg A \vee \wedge B)$ und so weiter.



3. Ein klassisches Perzeptron konvergiert nur, wenn der Datensatz linear separierbar ist, und es ist nicht in der Lage, die Wahrscheinlichkeiten für Kategorien abzuschätzen. Ein Klassifikator mit logistischer Regression konvergiert dagegen im Allgemeinen sogar dann zu einer ausreichend guten Lösung, wenn der Datensatz nichtlinear separierbar ist, und gibt Wahrscheinlichkeiten aus. Wenn Sie die Aktivierungsfunktion eines Perzeptrons durch die Sigmoid-Aktivierungsfunktion ersetzen (oder bei mehreren Neuronen die Softmax-Aktivierungsfunktion) und es mit dem Gradientenverfahren trainieren (oder einem anderen Optimierungsalgorithmus, der eine Kostenfunktion wie die Kreuzentropie minimiert), ist das Netz zur Klassifikation mit logistischer Regression äquivalent.
4. Die Sigmoid-Aktivierungsfunktion war ein Hauptbestandteil beim Trainieren der ersten MLPs, da ihre Ableitung immer ungleich null ist, sodass das Gradientenverfahren stets bergab rollen kann. Wenn die Aktivierungsfunktion eine Stufenfunktion ist, kann sich das Gradientenverfahren nicht bewegen, da es überhaupt keine Steigung gibt.
5. Häufig eingesetzte Aktivierungsfunktionen sind die Stufenfunktion, die Sigmoid-Funktion, der Tangens hyperbolicus und die Rectified Linear Unit (siehe Abbildung 10-8). Weitere Beispiele wie ELU und Variationen der ReLU finden Sie in Kapitel 11.
6. Für das in der Frage beschriebene MLP nehmen wir an, es bestünde aus einer Eingabeschicht mit 10 Neuronen, gefolgt von einer verborgenen Schicht mit 50 künstlichen Neuronen und schließlich einer Ausgabeschicht mit 3 künstlichen Neuronen. Alle künstlichen Neuronen verwenden ReLU als Aktivierungsfunktion.
   - Die Abmessungen der Eingabematrix $\mathbf{X}$ betragen $m \times 10$, wobei m für die Größe des Trainingsbatchs steht.
   - Die Abmessungen des Gewichtsvektors der verborgenen Schicht $\mathbf{W}_h$ betragen 10×50, und die Länge des Bias-Vektors $\mathbf{b}_h$ beträgt 50.
   - Die Abmessungen des Gewichtsvektors der Ausgabeschicht $\mathbf{W}_o$ betragen 50×3, und die Länge ihres Bias-Vektors $\mathbf{b}_o$ beträgt 3.
   - Die Abmessungen der Ausgabematrix des Netzes $\mathbf{Y}$ betragen $m \times 3$.

   - $\mathbf{Y} = \text{ReLU}(\text{ReLU}(\mathbf{X} \cdot \mathbf{W}_h + \mathbf{b}_h) \cdot \mathbf{W}_o + \mathbf{b}_o)$. Die ReLU-Funktion setzt einfach jeden negativen Wert in der Matrix auf null. Außerdem wird der Bias-Vektor beim Addieren zu einer Matrix zu jeder einzelnen Zeile der Matrix addiert. Dies bezeichnet man als *Broadcasting*.
7. Um E-Mails als Spam oder Ham zu klassifizieren, benötigen Sie nur ein Neuron in der Ausgabeschicht eines neuronalen Netzes – das beispielsweise die Wahrscheinlichkeit für Spam anzeigt. Sie würden in der Ausgabeschicht dazu normalerweise die Sigmoid-Aktivierungsfunktion verwenden. Wenn Sie stattdessen die MNIST-Aufgabe bearbeiten, benötigen Sie zehn Neuronen in der Ausgabeschicht und müssten die logistische Funktion durch die Softmax-Funktion ersetzen, die mit mehreren Kategorien umgehen kann. Dabei erhalten Sie eine Wahrscheinlichkeit pro Kategorie. Wenn Ihr neuronales Netz wie in Kapitel 2 Immobilienpreise vorhersagen soll, benötigen Sie ein Ausgabeneuron und überhaupt keine Aktivierungsfunktion in der Ausgabeschicht.

Wenn sich die vorherzusagenden Werte um viele Größenordnungen unterscheiden, sollten Sie statt der Zielgröße den Logarithmus der Zielgröße vorhersagen. Durch Berechnen der Exponentialfunktion aus der Ausgabe des Netzes erhalten Sie den geschätzten Wert (da $\exp(\log v) = v$).

8. Backpropagation ist ein Verfahren zum Trainieren künstlicher neuronaler Netze. Es berechnet zunächst die Gradienten der Kostenfunktion nach jedem Modellparameter (alle Gewichte und Biase) und führt dann einen Schritt im Gradientenverfahren mit diesen Gradienten aus. Dieser Schritt wird bei der Backpropagation normalerweise tausend- oder millionenfach ausgeführt, bis die Modellparameter zu Werten konvergieren, die die Kostenfunktion (hoffentlich) minimieren. Zur Berechnung der Gradienten verwendet das Backpropagation-Verfahren Autodiff im Reverse-Modus (auch wenn es bei der Einführung des Backpropagation-Verfahrens noch nicht so genannt wurde, es ist seitdem mehrmals neu erfunden worden). Autodiff im Reverse-Modus schreitet den Berechnungsgraphen vorwärts ab und berechnet den Wert jedes Knotens für den aktuellen Trainingsbatch und schreitet anschließend den Graphen rückwärts ab, wobei sämtliche Gradienten gleichzeitig berechnet werden (Details siehe Anhang B). Was ist also der Unterschied? Mit Backpropagation ist der gesamte Trainingsprozess eines künstlichen neuronalen Netzes über mehrere Backpropagation-Schritte gemeint, wobei in jedem einzelnen Gradienten berechnet und ein Schritt im Gradientenverfahren durchgeführt wird. Im Gegensatz dazu ist Autodiff im Reverse-Modus einfach eine Technik zum effizienten Berechnen von Gradienten, die zufällig vom Backpropagation-Verfahren verwendet wird.
9. Hier folgt eine Liste aller Hyperparameter, die Sie in einem einfachen MLP verändern können: die Anzahl verborgener Schichten, die Anzahl Neuronen pro verborgene Schicht und die in jeder Schicht verwendete Aktivierungsfunktion.

Im Allgemeinen ist die Aktivierungsfunktion ReLU (oder eine ihrer Varianten, siehe Kapitel 11) eine gute Standardeinstellung für die verborgenen Schichten. In der Ausgabeschicht sollten Sie bei binärer Klassifikation die Sigmoid-Aktivierungsfunktion verwenden, bei mehreren Kategorien die Softmax-Aktivierungsfunktion und bei Regression überhaupt keine Aktivierungsfunktion. Wenn das MLP die Trainingsdaten overfittet, können Sie die Anzahl verborgener Schichten und die Anzahl der Neuronen darin reduzieren.

Die Lösung finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 11: Trainieren von Deep-Learning-Netzen

1. Die Initialisierungen nach Glorot und He wurden entwickelt, um die Ausgabestandardabweichung so nah wie möglich an der Eingabestandardabweichung zu halten – zumindest zu Beginn des Trainings. Das verringert das Problem der verschwindenden oder explodierenden Gradienten.
2. Nein, alle Gewichte sollten unabhängig voneinander erzeugt werden; es sollten nicht alle den gleichen Startwert haben. Ein wichtiges Ziel beim zufälligen Erzeugen von Gewichten ist, Symmetrien aufzubrechen: Wenn alle Gewichte den gleichen Startwert haben, selbst wenn dieser ungleich null ist, besteht eine Symmetrie (d. h., sämtliche Neuronen einer Schicht sind äquivalent). Das Backpropagation-Verfahren ist nicht in der Lage, diese aufzubrechen. Anders ausgedrückt: Alle Neuronen mit gleichen Gewichten verhalten sich innerhalb einer Schicht wie ein einziges Neuron, sie sind nur viel langsamer. Es ist praktisch unmöglich, dass eine derartige Anordnung zu einer guten Lösung konvergiert.
3. Es ist völlig in Ordnung, die Bias-Terme mit null zu initialisieren. Manchmal werden sie genau wie die Gewichte initialisiert, auch das ist in Ordnung; es macht keinen großen Unterschied.
4. ReLU ist im Allgemeinen eine gute Standardwahl für verborgene Schichten, da es schnell ist und zu guten Ergebnissen führt. Die Tatsache, dass die ReLU-Aktivierungsfunktion exakt null ausgeben kann (siehe zum Beispiel Kapitel 17), ist ebenfalls in manchen Fällen nützlich. Zudem kann es manchmal von optimierten Implementierungen und von Hardwarebeschleunigung profitieren. Die Leaky-ReLU-Varianten von ReLU können die Qualität des Modells verbessern, ohne seine Geschwindigkeit im Vergleich zu ReLU allzu sehr zu verringern. Für große neuronale Netze und komplexere Probleme können GLU, Swish und Mish ein qualitativ etwas besseres Modell liefern, aber auf Kosten erhöhter Rechenleistung. Der Tangens hyperbolicus (tanh) ist in der Ausgabeschicht nützlich, wenn Sie eine Zahl in einem definierten Bereich (standardmäßig zwischen –1 und 1) ausgeben möchten, er wird aber heutzutage in verborgenen Schichten kaum noch verwendet, außer in rekurrenten Netzen. Die Sigmoid-Aktivierungsfunktion ist ebenfalls in der Ausgabeschicht

nützlich, wenn Sie eine Wahrscheinlichkeit schätzen möchten (z. B. bei der binären Klassifikation). Auch sie findet selten in verborgenen Schichten Anwendung (es gibt Ausnahmen – beispielsweise in der codierenden Schicht eines Variational Autoencoder, siehe Kapitel 17). Die Softplus-Aktivierungsfunktion wird in der Ausgabeschicht verwendet, wenn Sie sicherstellen müssen, dass die Ausgabe immer positiv ist. Die Softmax-Aktivierungsfunktion ist in Ausgabeschichten nützlich, um Wahrscheinlichkeiten sich gegenseitig ausschließender Kategorien abzuschätzen. Auch sie wird selten (falls überhaupt) in verborgenen Schichten eingesetzt.

5. Wenn Sie bei einem `SGD`-Optimierer den Hyperparameter `momentum` zu nah an 1 setzen (z. B. 0,99999), wird der Algorithmus voraussichtlich stark an Geschwindigkeit aufnehmen, sich hoffentlich in etwa auf das globale Minimum zubewegen und dann daran vorbeizischen. Dann bremst er ab, kehrt zurück, beschleunigt wieder, fliegt erneut zu weit und so weiter. Bis zur Konvergenz kann er auf diese Weise viele Male oszillieren, das Konvergieren dauert also insgesamt viel länger als mit einem kleineren Wert für `momentum`.
6. Ein dünn besetztes Modell (bei dem die meisten Gewichte null betragen) lässt sich erzeugen, indem Sie das Modell normal trainieren und dann winzige Gewichte auf null setzen. Zusätzlich können Sie beim Trainieren die ℓ_1-Regularisierung anwenden, was den Optimierer in Richtung dünn besetzter Parameter drückt. Eine dritte Möglichkeit ist, das TensorFlow Optimization Toolkit einzusetzen.
7. Ja, das Trainieren wird durch Drop-out langsamer, meist etwa um den Faktor zwei. Es hat allerdings keinen Einfluss auf die Inferenzgeschwindigkeit, da Drop-out nur beim Trainieren angeschaltet ist. MC-Drop-out verhält sich beim Training genauso, aber es ist auch während der Inferenz aktiv, daher wird jede Inferenz ein bisschen verlangsamt. Entscheidender ist, dass Sie beim Einsatz von MC-Drop-out im Allgemeinen die Inferenz zehn Mal laufen lassen wollen, um bessere Vorhersagen zu erhalten. Das heißt, dass sich das Treffen von Vorhersagen um den Faktor 10 oder mehr verlangsamt.

Die Lösung zur Übungsaufgabe 8 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 12: Eigene Modelle und Training mit TensorFlow

1. TensorFlow ist eine Open-Source-Bibliothek für numerisches Rechnen, die besonders auf Machine Learning im großen Maßstab ausgelegt ist. Ihr Kern ähnelt NumPy, aber sie bietet zudem GPU-Unterstützung, sie ermöglicht verteiltes Rechnen, die Analyse von Rechengraphen und Optimierungsmöglichkeiten (mit einem portierbaren Graphenformat, das Ihnen erlaubt, ein TensorFlow-

Modell in einer Umgebung zu trainieren und in einer anderen auszuführen), eine Optimierungs-API, die auf Autodiff im Reverse-Mode basiert, und eine Reihe leistungsfähiger APIs wie tf.keras, tf.data, tf.image oder tf.signal. Andere beliebte Bibliotheken zum Deep Learning sind PyTorch, MXNet, Microsoft Cognitive Toolkit, Theano, Caffe2 und Chainer.

2. Auch wenn TensorFlow einen Großteil der Funktionalität von NumPy bietet, können Sie es nicht einfach austauschen: Die Namen der Funktionen sind nicht immer die gleichen (z. B. `tf.reduce_sum()` vs. `np.sum()`), manche Funktionen verhalten sich nicht genau gleich (so erzeugt `tf.transpose()` eine transponierte Kopie eines Tensors, während das Attribut `T` bei NumPy eine transponierte Sicht bietet, ohne tatsächlich Daten zu kopieren), und schließlich sind Arrays bei NumPy veränderbar, während das für die Tensoren bei TensorFlow nicht gilt (Sie können aber eine `tf.Variable` nutzen, wenn Sie ein veränderbares Objekt benötigen).
3. Sowohl `tf.range(10)` wie auch `tf.constant(np.arange(10))` geben einen eindimensionalen Tensor mit den Integer-Werten 0 bis 9 zurück. Aber beim ersten kommen 32-Bit-Ganzzahlen zum Einsatz, während im zweiten Fall 64-Bit-Werte genutzt werden. TensorFlow verwendet standardmäßig 32 Bits, während NumPy mit 64 Bits arbeitet.
4. Neben normalen Tensoren bietet TensorFlow noch viele andere Datenstrukturen, unter anderem Sparse-Tensoren, Tensor-Arrays, Ragged-Tensoren, Queues, String-Tensoren und Sets. Die letzten beiden werden sogar als normale Tensoren repräsentiert, aber TensorFlow bietet zusätzliche Funktionen zu ihrer Bearbeitung an (in `tf.strings` und `tf.sets`).
5. Wollen Sie eine eigene Verlustfunktion definieren, können Sie sie im Allgemeinen als normale Python-Funktion implementieren. Aber wenn sie Hyperparameter (oder einen anderen Status) unterstützen muss, sollten Sie eine Unterklasse von `keras.losses.Loss` erstellen und die Methoden `__init__()` und `call()` implementieren. Sollen die Hyperparameter der Verlustfunktion zusammen mit dem Modell gesichert werden, müssen Sie zudem die Methode `get_config()` implementieren.
6. Wie eigene Verlustfunktionen können die meisten Metriken als normale Python-Funktionen definiert werden. Soll Ihre Metrik auch Hyperparameter (oder einen anderen Status) unterstützen, erstellen Sie eine Unterklasse von `keras.metrics.Metric`. Ist das Berechnen der Metrik über eine ganze Epoche zudem nicht das Gleiche wie das Berechnen der mittleren Metrik über alle Batches in dieser Epoche (zum Beispiel bei Relevanz und Sensitivität), sollten Sie eine Unterklasse von `keras.metrics.Metric` bilden und die Methoden `__init__()`, `update_state()` und `result()` implementieren, um eine gleitende Metrik für jede Epoche zu erschaffen. Außerdem sollten Sie die Methode `reset_states()` implementieren, sofern nicht nur einfach alle Variablen auf 0,0 zu setzen sind. Wollen Sie den Status zusammen mit dem Modell abspeichern, sollten Sie auch noch die Methode `get_config()` implementieren.

7. Sie sollten die internen Komponenten Ihres Modells (also die Schichten oder wiederverwendbaren Blöcke mit Schichten) vom Modell selbst trennen (also das zu trainierende Objekt). Erstere sollten Unterklassen von `keras.layers.Layer` sein, Letzteres eine Unterklasse von `keras.models.Model`.
8. Es ist schon recht fortgeschritten, eine eigene Trainingsschleife zu schreiben, daher sollten Sie das nur in Angriff nehmen, wenn Sie es wirklich brauchen. Keras stellt diverse Werkzeuge zum Anpassen des Trainings bereit, ohne dass Sie eine eigene Trainingsschleife schreiben müssen: Callbacks, eigene Regularisierer, Constraints, Verluste und so weiter. Sie sollten diese wann immer möglich verwenden: Durch das Schreiben einer eigenen Trainingsschleife fangen Sie sich eher Fehler ein, außerdem wird sich der Code schlechter wiederverwenden lassen. Aber manchmal ist das Schreiben einer eigenen Trainingsschleife notwendig – wenn Sie zum Beispiel verschiedene Optimierer für unterschiedliche Teile Ihres neuronalen Netzes verwenden wollen (wie im Wide-&-Deep-Artikel (*https://homl.info/widedeep*)). Sie kann auch beim Debuggen nützlich sein oder wenn Sie verstehen wollen, wie das Training genau abläuft.

   Eigene Keras-Komponenten sollten in TF Functions konvertierbar sein – Sie sollten sich also so weit wie möglich an TF-Operationen und alle in Kapitel 12 aufgeführten Regeln (Abschnitt »*Regeln für TF Functions*«) halten. Müssen Sie unbedingt anderen Python-Code in einer eigenen Komponente verwenden, können Sie ihn entweder in einer `tf.py_function()`-Operation verpacken (das wird aber die Performance verringern und die Portierbarkeit Ihres Modells einschränken) oder beim Erstellen der eigenen Schicht oder des Modells `dynamic=True` setzen (oder die Modellmethode `compile()` mit `run_eagerly=True` aufrufen).
9. In Kapitel 12 finden Sie im Abschnitt »Regeln für TF Functions« die Liste mit den Regeln für das Erstellen einer TF Function.
10. Es kann zum Debuggen nützlich sein, ein dynamisches Keras-Modell zu erstellen, da dann keine eigenen Komponenten in eine TF Function umgewandelt werden und Sie Ihren Code mit einem beliebigen Python-Debugger untersuchen können. Außerdem ist es praktisch, wenn Sie beliebigen Python-Code oder auch Aufrufe externer Bibliotheken in Ihrem Modell (oder im Trainingscode) einsetzen wollen. Um ein Modell dynamisch zu machen, müssen Sie beim Erstellen `dynamic=True` setzen. Alternativ können Sie `run_eagerly=True` setzen, wenn Sie die Methode `compile()` des Modells aufrufen. Wenn Sie ein Modell dynamisch machen, halten Sie Keras davon ab, TensorFlow-Graphen zu verwenden. Damit werden Training und Inferenz langsamer, und Sie haben nicht die Möglichkeit, den Rechengraphen zu exportieren, wodurch Sie die Portierbarkeit Ihres Modells einschränken.

Die Lösungen zu den Übungsaufgaben 12 und 13 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 13: Daten mit TensorFlow laden und vorverarbeiten

1. Das Einlesen eines großen Datensatzes und seine effiziente Vorverarbeitung können für Entwicklerinnen und Entwickler eine komplexe Aufgabe sein. Die Data-API erleichtert das sehr. Sie bietet viele Features, unter anderem das Laden der Daten aus unterschiedlichen Quellen (wie zum Beispiel Text- oder Binärdateien), das parallele Lesen aus mehreren Quellen, das Umwandeln und Verweben der Datensätze, ein Durchmischen, in Batches einteilen und das Prefetchen von Daten.
2. Durch das Aufteilen eines großen Datensatzes auf mehrere Dateien können Sie die Daten grober vormischen, bevor Sie sie dann mit einem Shuffling Buffer feiner durchmischen. Zudem können Sie so mit riesigen Datenmengen umgehen, die nicht in einen einzelnen Rechner passen. Es ist auch einfacher, Tausende kleiner Dateien zu verarbeiten als eine riesige Datei – die Daten lassen sich so beispielsweise in mehrere Untermengen aufteilen. Und wenn die Daten über mehrere Dateien auf mehreren Servern verteilt sind, ist es zudem noch möglich, sie simultan herunterzuladen, was die verfügbare Bandbreite besser auslastet.
3. Sie können mit TensorBoard Profiling-Daten visualisieren. Wird die GPU nicht vollständig ausgelastet, wird vermutlich Ihre Eingabepipeline der Flaschenhals sein. Das können Sie beheben, indem Sie die Daten parallel in mehreren Threads lesen, dann vorverarbeiten und sicherstellen, dass ein paar Batches prefetcht werden. Wenn das nicht ausreicht, damit Ihre GPU während des Trainings 100 % Last hat, achten Sie darauf, dass der Code zum Vorverarbeiten optimiert ist. Sie können auch versuchen, den Datensatz in mehreren TFRecord-Dateien zu sichern und eventuell einen Teil der Vorverarbeitung schon im Voraus zu erledigen, sodass es nicht während des Trainings geschehen muss (hier kann TF Transform helfen). Verwenden Sie nötigenfalls eine Maschine mit mehr CPU und RAM und stellen Sie sicher, dass die GPU-Bandbreite ausreicht.
4. Eine TFRecord-Datei besteht aus einer Folge beliebiger Binärdatensätze: Sie können wirklich jegliche Binärdaten in jedem Datensatz ablegen. Aber in der Praxis enthalten die meisten TFRecord-Dateien Sequenzen serialisierter Protocol Buffer. Damit können Sie die Vorteile dieser Protocol Buffer nutzen, wie zum Beispiel, dass sie sich leicht auf mehreren Plattformen und mit vielen Programmiersprachen lesen lassen, zudem können ihre Definitionen auch im Nachhinein noch abwärtskompatibel geändert werden.
5. Das Protobuf-Format `Example` bietet den Vorteil, dass TensorFlow Operationen zum Parsen anbietet (die `tf.io.parse*example()`-Funktionen), ohne dass Sie Ihr eigenes Format definieren müssen. Es ist für die meisten Datensatzin-

stanzen ausreichend flexibel. Aber wenn Ihr Anwendungsfall nicht abgedeckt ist, können Sie Ihren eigenen Protocol Buffer definieren, ihn mit `protoc` kompilieren (die Argumente `--descriptor_set_out` und `--include_imports` sorgen für einen Export des Protobuf-Deskriptors) und mit der Funktion `tf.io.decode_proto()` die serialisierten Protobufs parsen (im Abschnitt »Custom protobuf« des Notebooks finden Sie ein Beispiel). Das Ganze ist komplizierter und erfordert das Deployen des Deskriptors zusammen mit dem Modell, aber es ist machbar.

6. Beim Einsatz von TFRecords werden Sie normalerweise die Komprimierung aktivieren wollen, wenn die TFRecord-Dateien vom Trainingsskript heruntergeladen werden müssen, da die Kompression die Dateien verkleinert und damit die Zeit zum Herunterladen verringert. Befinden sich die Dateien aber auf der gleichen Maschine wie das Trainingsskript, ist es im Allgemeinen besser, die Kompression abzuschalten, um die CPU nicht unnötig zu belasten.
7. Schauen wir uns die Vor- und Nachteile jeder Vorverarbeitungsmöglichkeit an:
   - Verarbeiten Sie die Daten beim Erstellen der Datendateien vor, wird das Trainingsskript schneller laufen, da dann nicht beim Training vorverarbeitet werden muss. In manchen Fällen werden die vorverarbeiteten Daten auch viel kleiner als die ursprünglichen Daten sein, dann können Sie so Platz sparen und ein Herunterladen beschleunigen. Es kann auch hilfreich sein, die vorverarbeiteten Daten greifbar zu haben, um sie beispielsweise zu untersuchen oder zu archivieren. Aber es gibt auch Nachteile. Es ist zum Beispiel nicht so einfach, mit mehreren Vorverarbeitungslogiken zu experimentieren, wenn Sie einen vorverarbeiteten Datensatz für jede Variante benötigen. Und wenn Sie eine Data Augmentation durchführen wollen, kann es sein, dass Sie viele Varianten Ihres Datensatzes vorhalten müssen, die viel Plattenplatz benötigen und viel Zeit zum Generieren brauchen. Schließlich wird das trainierte Modell auch vorverarbeitete Daten erwarten, und Sie werden Ihre Anwendung um Code zum Vorverarbeiten ergänzen müssen, bevor diese das Modell aufruft. Bei diesem Vorgehen besteht das Risiko doppelten Codes und eines Preprocessing Mismatch.
   - Werden die Daten mit der tf.data-Pipeline vorverarbeitet, ist es viel einfacher, die Vorverarbeitungslogik anzupassen und Data Augmentation umzusetzen. Zudem macht es tf.data einfach, sehr effiziente Vorverarbeitungspipelines umzusetzen (zum Beispiel mit Multithreading und Prefetching). Aber das Vorverarbeiten der Daten wird das Training verlangsamen. Und jede Trainingsinstanz wird in jeder Epoche vorverarbeitet statt nur einmal beim Erstellen der Datendateien – es sei denn, der Datensatz passt in den Speicher und Sie können ihn mit der Methode `cache()` des Datasets cachen. Schließlich wird das trainierte Modell immer noch vorverarbeitete Daten erwarten. Nutzen Sie in Ihrer tf.data-Pipeline Vorverarbeitungsschichten für den Vorverarbeitungsschritt, können Sie diese

Schichten in Ihrem finalen Modell wiederverwenden (indem Sie sie nach dem Training hinzufügen), um doppelten Code und ein Preprocessing Mismatch zu vermeiden.

- Ergänzen Sie Ihr Modell um Schichten zur Vorverarbeitung, müssen Sie den Code zum Vorverarbeiten für Training und Inferenz nur einmal schreiben. Muss Ihr Modell auf vielen verschiedenen Plattformen deployt werden, muss der Code zur Vorverarbeitung nicht mehrfach geschrieben werden. Zudem laufen Sie nicht Gefahr, die falsche Vorverarbeitungslogik für Ihr Modell zu nutzen, da sie Teil des Modells ist. Andererseits wird ein Vorverarbeiten der Daten während des Trainings alles langsamer machen, und jede Instanz wird einmal pro Epoche vorverarbeitet.

8. Schauen wir uns an, wie man kategorische Textmerkmale und Text codieren kann:
   - Um ein kategorisches Merkmal mit einer natürlichen Reihenfolge zu codieren, wie zum Beispiel eine Filmbewertung (»schlecht«, »mittelmäßig«, »gut«), ist die einfachste Option eine Ordinalcodierung: Sortieren Sie die Kategorien nach ihrer natürlichen Reihenfolge und bilden Sie jede auf ihren Rang ab (»schlecht« wird zu 0, »mittelmäßig« zu 1 und »gut« zu 2). Aber die meisten kategorischen Merkmale haben keine solche natürliche Reihenfolge. Es gibt zum Beispiel keine für Berufe oder Länder. In diesem Fall können Sie bei vielen Kategorien eine One-Hot-Codierung oder Embeddings einsetzen. Bei Keras kann die `StringLookup`-Schicht für ordinales Codieren (mit dem standardmäßigen `output_mode="int"`) oder One-Hot-Codierung (mit `output_mode="one_hot"`) genutzt werden. Auch eine Multi-Hot-Codierung ist möglich (mit `output_mode="multi_hot"`), wenn Sie mehrere kategorische Textmerkmale gemeinsam codieren wollen – sofern sie die gleichen Kategorien teilen und es egal ist, welches Merkmal zu welcher Kategorie beigetragen hat. Bei trainierbaren Embeddings müssen Sie erst die `StringLookup`-Schicht nutzen, um eine ordinale Codierung zu erreichen, und können danach die Embedding-Schicht verwenden.
   - Bei Text lässt sich die `TextVectorization`-Schicht leicht einsetzen, und für einfache Aufgaben kann das auch gut funktionieren – oder Sie nutzen TF Text für weitergehende Features. Aber oft werden Sie vortrainierte Sprachmodelle einsetzen wollen, die Sie über Tools wie TF Hub oder die Transformers-Bibliothek von Hugging Face erhalten können. Diese zwei letzten Optionen werden in Kapitel 16 behandelt.

Die Lösungen zu den Übungsaufgaben 9 und 10 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 14: Deep Computer Vision mit Convolutional Neural Networks

1. Folgendes sind die Hauptvorteile eines CNN gegenüber einem vollständig verbundenen DNN zur Bildklassifikation:
   - Weil aufeinanderfolgende Schichten nur teilweise verbunden sind und Gewichte in großem Ausmaß wiederverwendet werden, enthält ein CNN viel weniger Parameter als ein vollständig verbundenes DNN, wodurch es schneller trainierbar ist, weniger zu Overfitting neigt und viel weniger Trainingsdaten erfordert.
   - Wenn ein CNN einen Kernel erlernt hat, der ein bestimmtes Muster erkennt, kann es dieses Muster überall im Bild erkennen. Im Gegensatz dazu kann ein DNN, wenn es ein Muster an einem Ort erlernt hat, dieses nur am gleichen Ort wiedererkennen. Da Bilder meist wiederkehrende Muster enthalten, können CNNs bei der Verarbeitung von Bilddaten sehr viel besser und mit weniger Trainingsbeispielen als DNNs verallgemeinern, beispielsweise bei der Klassifikation.
   - Schließlich verfügt ein DNN über keinerlei Wissen darüber, wie Pixel organisiert sind; es weiß nicht, dass benachbarte Pixel im Bild nah beieinanderliegen. Die Architektur eines CNN hat diese Art Vorwissen verinnerlicht. Die niedrigeren Schichten erkennen für gewöhnlich Muster in kleinen Bildausschnitten, die oberen Schichten kombinieren dagegen die Muster der unteren Schichten zu größeren Mustern. Dies funktioniert mit den meisten natürlichen Bildern gut, womit sich CNNs einen guten Vorsprung vor DNNs erarbeiten.
2. Rechnen wir aus, wie viele Parameter das CNN hat. Der erste Convolutional Layer enthält 3 × 3 Kernels, die Eingabe weist drei Kanäle auf (Rot, Grün und Blau), jede Feature Map enthält 3 × 3 × 3 Gewichte sowie einen Bias-Term. Damit ergeben sich 28 Parameter pro Feature Map. Da dieser erste Convolutional Layer 100 Feature Maps enthält, gibt es insgesamt 2.800 Parameter. Der zweite Convolutional Layer enthält 3 × 3 Kernels, und die Eingabe sind die 100 Feature Maps aus der vorherigen Schicht. Damit enthält jede Feature Map 3 × 3 × 100 = 900 Gewichte sowie einen Bias-Term. Da es 200 Feature Maps gibt, enthält diese Schicht 901 × 200 = 180.200 Parameter. Der dritte und letzte Convolutional Layer enthält ebenfalls 3 × 3 Kernels, und seine Eingabe sind die 200 Feature Maps aus den vorherigen Schichten. Also enthält jede Feature Map 3 × 3 × 200 = 1.800 Gewichte sowie einen Bias-Term. Da es 400 Feature Maps gibt, enthält diese Schicht insgesamt 1.801 × 400 = 720.400 Parameter. Rechnet man alles zusammen, hat dieses CNN 2.800 + 180.200 + 720.400 = 903.400 Parameter.

   Rechnen wir nun aus, wie viel RAM dieses neuronale Netz (mindestens) beim Treffen einer Vorhersage für einen Datenpunkt benötigt. Dazu rechnen wir

zunächst die Größen der Feature Maps in jeder Schicht aus. Da wir als Stride 2 und `"same"`-Padding verwenden, werden die horizontale und die vertikale Dimension jeder Feature Map in jeder Schicht durch 2 geteilt (aufgerundet, falls nötig). Da also die Eingabekanäle aus 200 × 300 Pixeln bestehen, haben die Feature Maps in der ersten Schicht die Größe 100 × 150, die Feature Maps in der zweiten Schicht die Größe 50 × 75 und die Feature Maps in der dritten Schicht die Größe 25 × 38. Da 32 Bits 4 Bytes entsprechen und der erste Convolutional Layer aus 100 Feature Maps besteht, nimmt die erste Schicht 4 × 100 × 150 × 100 = 6 Millionen Bytes ein (6 MB). Die zweite Schicht benötigt 4 × 50 × 75 × 200 = 3 Millionen Bytes (3 MB). Schließlich benötigt die dritte Schicht 4 × 25 × 38 × 400 = 1.520.000 Bytes (1,5 MB). Sobald eine Schicht berechnet wurde, kann der von der vorherigen Schicht verwendete Speicher freigegeben werden, bei guter Optimierung sind also im RAM nur 6 + 3 = 9 Millionen Bytes (9 MB) nötig (wenn die zweite Schicht soeben fertig berechnet ist, der von der ersten Schicht belegte Speicher aber noch nicht freigegeben wurde). Einen Moment! Wir müssen auch den von den Parametern des CNN belegten Speicher berücksichtigen. Wir haben oben 903.400 Parameter ausgerechnet, von denen jeder 4 Bytes belegt. Dadurch kommen noch einmal 3.613.600 Bytes hinzu (etwa 3,6 MB). Das gesamte nötige RAM beträgt (mindestens) 12.613.600 Bytes (etwa 12,6 MB).

Als Letztes berechnen wir die Mindestgröße des benötigten RAM beim Trainieren des CNN mit einem Mini-Batch aus 50 Bildern. Beim Trainieren verwendet TensorFlow das Backpropagation-Verfahren, bei dem sämtliche im Vorwärtsdurchlauf berechneten Werte aufgehoben werden müssen, bis der Rückwärtsdurchlauf beginnt. Daher müssen wir das gesamte von allen Schichten benötigte RAM für einen Datenpunkt berechnen und diesen Wert mit 50 multiplizieren! An dieser Stelle rechnen wir in Megabytes statt in Bytes weiter. Wir hatten oben berechnet, dass die drei Schichten jeweils 6, 3 und 1,5 MB pro Datenpunkt benötigen. Das sind insgesamt 10,5 MB pro Datenpunkt. Bei 50 Datenpunkten benötigen wir also 525 MB an RAM. Dazu kommt das für die Eingabebilder benötigte RAM, also 50 × 4 × 200 × 300 × 3 = 36 Millionen Bytes (36 MB) sowie das für die Modellparameter benötigte RAM, die oben berechneten 3,6 MB. Das für die Gradienten nötige RAM ignorieren wir an dieser Stelle, da es im Verlauf des Backpropagation-Verfahrens im Rückwärtsdurchlauf nach und nach freigegeben wird). Wir erhalten insgesamt etwa 525 + 36 + 3,6 = 564,6 MB. Und das ist wirklich nur eine optimistisch errechnete untere Grenze.

3. Wenn Ihrer GPU beim Trainieren eines CNN der Speicher ausgeht, können Sie folgende fünf Dinge tun, um das Problem zu bekämpfen (oder eine GPU mit mehr RAM kaufen):
   - Die Größe der Mini-Batches verringern.
   - Die Dimensionalität verringern, indem Sie in einer oder mehreren Schichten einen größeren Stride einstellen.
   - Eine oder mehrere Schichten entfernen.

   - Floats mit 16 Bit anstatt mit 32 Bit verwenden.
   - Das CNN über mehrere Geräte verteilen.
4. Eine Max-Pooling-Schicht enthält überhaupt keine Parameter, wohingegen ein Convolutional Layer recht viele enthält (siehe vorherige Fragen).
5. Ein Local Response Normalization Layer sorgt dafür, dass die am stärksten aktivierten Neuronen die Neuronen an der gleichen Position in benachbarten Feature Maps hemmen, wodurch die Feature Maps dazu angehalten werden, sich unterschiedlich zu entwickeln und sich eine größere Bandbreite an Mustern anzueignen. Er wird normalerweise in den unteren Schichten eingesetzt, um einen größeren Pool kleinteiliger Merkmale zu erhalten, auf den die nachgeschalteten Schichten aufbauen können.
6. Die wichtigsten Innovationen bei AlexNet gegenüber LeNet-5 sind, dass es wesentlich größer und tiefer ist und Convolutional Layers direkt aufeinanderstapelt, anstatt auf jedem Convolutional Layer einen Pooling Layer zu platzieren. Die wichtigste Innovation bei GoogLeNet ist die Einführung von *Inception-Modulen*, die ein weitaus tieferes Netz als bei früheren CNN-Architekturen bei weniger Parametern ermöglichen. Die wichtigste Neuerung von ResNet sind die Skip-Verbindungen, mit denen mehr als 100 Schichten möglich sind. Auch seine Einfachheit und seine Konsistenz können als innovativ gelten. Der wichtigste Fortschritt von SENet war die Idee, einen SE-Block (ein Dense-Netz aus zwei Schichten) nach jedem Inception-Modul in einem Inception-Netz oder jeder Residual Unit in einem ResNet zu verwenden, um die relative Wichtigkeit von Feature Maps neu zu kalibrieren. Die wichtigste Innovation von Xception war der Einsatz der Depthwise Separable Convolutional Layers, die sich räumliche Muster und Muster entlang der Tiefe getrennt anschauen. Bei EfficientNet war schließlich die Compound-Scaling-Methode am wichtigsten, durch die ein Modell effizient auf größere Rechenleistung skaliert werden kann.
7. Fully Convolutional Networks sind neuronale Netze, die nur aus Convolutional und Pooling Layers bestehen. FCNs können Bilder beliebiger Breite und Höhe (zumindest oberhalb einer Minimalgröße) effizient verarbeiten. Sie sind für die Objekterkennung und semantische Segmentierung am nützlichsten, weil sie sich das Bild nur einmal anschauen müssen (statt ein CNN mehrfach über die verschiedenen Teile des Bilds laufen zu lassen). Haben Sie ein CNN mit Dense-Schichten obendrauf, können Sie diese Dense-Schichten in Convolutional Layers umwandeln, um ein FCN zu schaffen: Ersetzen Sie einfach die unterste Dense-Schicht durch einen Convolutional Layer mit einer Kernelgröße gleich der Eingabegröße der Schicht, mit einem Filter pro Neuron in der Dense-Schicht und dem Einsatz des `"valid"`-Padding. Die Schrittweite sollte im Allgemeinen 1 sein, aber Sie können sie auf einen höheren Wert setzen, wenn Sie möchten. Die Aktivierungsfunktion sollte die gleiche wie die der Dense-Schicht sein. Die anderen Dense-Schichten sollten genauso konvertiert

werden, allerdings mit 1 × 1-Filtern. Es ist sogar möglich, ein trainiertes CNN auf diese Art und Weise umzuwandeln, indem die Gewichtsmatrizen der Dense-Schichten passend umgeformt werden.

8. Die größte technische Schwierigkeit einer semantischen Segmentierung liegt darin, dass in einem CNN viel räumliche Auflösung verloren geht, wenn das Signal die Schichten durchläuft, insbesondere in Pooling Layers und in Schichten mit einer Schrittweite größer 1. Diese räumliche Information muss irgendwie wiederhergestellt werden, um die Kategorie jedes Pixels genau vorhersagen zu können.

Die Lösungen zu den Übungsaufgaben 9 bis 12 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 15: Verarbeiten von Sequenzen mit RNNs und CNNs

1. Dies sind einige Anwendungsgebiete von RNNs:
   - Bei einem Sequence-to-Sequence-RNN: Wettervorhersage (oder Vorhersage einer beliebigen Zeitreihe), maschinelle Übersetzung (mit einer Encoder-Decoder-Architektur), Videos mit Untertiteln versehen, Umwandlung von Sprache zu Text, Erzeugen von Musik (oder anderen Sequenzen) und Identifizieren der Akkorde in einem Musikstück.
   - Bei einem Sequence-to-Vector-RNN: Klassifizieren von Musikstücken nach Genre, Analysieren der Meinung einer Buchrezension, aus den Signalen von Gehirnimplantaten vorhersagen, an welches Wort ein aphasischer Patient denkt, aus bereits gesehenen Videos die Wahrscheinlichkeit vorherzusagen, mit der ein Nutzer einen bestimmten Film sehen möchte (dies ist eine von vielen möglichen Implementierungen von *kollaborativen Filtern* für ein Empfehlungssystem).
   - Bei einem Vector-to-Sequence-RNN: Bilder mit Untertiteln versehen, eine Playlist von Musikstücken aus einem Embedding des aktuellen Interpreten erstellen, eine Melodie anhand einer Parameterliste erzeugen, Fußgänger auf einem Bild erkennen (z. B. einer Momentaufnahme der Kamera eines selbstfahrenden Autos).
2. Eine RNN-Schicht muss dreidimensionale Eingaben haben: Die erste Dimension ist die Batchdimension (dessen Größe die Batchgröße ist), die zweite repräsentiert die Zeit (deren Größe die Anzahl an Zeitschritten ist), und die dritte enthält die Eingaben für jeden Zeitschritt (deren Größe die Anzahl an Eingabemerkmalen pro Zeitschritt ist). Wollen Sie zum Beispiel einen Batch mit fünf Zeitserien mit jeweils zehn Zeitschritten und zwei Werten pro Zeitschritt verarbeiten (etwa die Temperatur und die Windgeschwindigkeit), ist die Form [5, 10, 2]. Die Ausgaben sind ebenfalls dreidimensional: Die ersten

beiden Dimensionen sind gleich, die dritte entspricht der Zahl von Neuronen. Verarbeitet beispielsweise eine RNN-Schicht mit 32 Neuronen den eben besprochenen Batch, wird die Ausgabe die Form [5, 10, 32] haben.

3. Um ein tiefes Sequence-to-Sequence-RNN mit Keras zu bauen, müssen Sie für alle RNN-Schichten `return_sequences=True` setzen. Bei einem Sequence-to-Vector-RNN setzen Sie `return_sequences=True` für alle RNN-Schichten mit Ausnahme der obersten, bei der `return_sequences=False` stehen muss (oder Sie setzen gar nichts, da `False` der Standardwert ist).
4. Haben Sie eine tägliche univariate Zeitserie und wollen Sie die nächsten sieben Tage vorhersagen, ist die einfachste RNN-Architektur dafür ein Stapel RNN-Schichten (alle mit `return_sequences=True` außer für die oberste RNN-Schicht) und sieben Neuronen in der Ausgabe-RNN-Schicht. Dann können Sie dieses Modell mit zufälligen Fenstern aus der Zeitserie trainieren (zum Beispiel Sequenzen mit 30 aufeinanderfolgenden Tagen als Eingabe und einem Vektor mit den Werten der nächsten 7 Tage als Ziel). Das ist ein Sequence-to-Vector-RNN. Alternativ könnten Sie `return_sequences=True` für alle RNN-Schichten setzen, um ein Sequence-to-Sequence-RNN zu erstellen. Dann trainieren Sie das Modell mit zufälligen Fenstern aus der Zeitserie mit Sequenzen, die so lang wie die Ziele sind. Jede Zielsequenz sollte sieben Werte pro Zeitschritt enthalten (zum Beispiel sollte das Ziel für Zeitschritt t ein Vektor mit den Werten an den Zeitschritten $t + 1$ bis $t + 7$ sein).
5. Die zwei größten Probleme beim Trainieren von RNNs sind instabile Gradienten (explodierend oder verschwindend) und ein sehr beschränktes Kurzzeitgedächtnis. Diese Probleme werden beim Umgang mit langen Sequenzen noch schlimmer. Um die instabilen Gradienten im Griff zu behalten, können Sie eine kleinere Lernrate nutzen, auf eine sättigende Aktivierungsfunktion wie tanh zurückgreifen (den Standard) und eventuell bei jedem Zeitschritt Gradient Clipping, Layer Normalization oder Drop-out einsetzen. Um das beschränkte Kurzzeitgedächtnis anzugehen, können Sie `LSTM`- oder `GRU`-Schichten nutzen (das hilft ebenfalls bei instabilen Gradienten).
6. Die Architektur einer LSTM-Zelle sieht kompliziert aus, aber eigentlich ist sie gar nicht so schwer zu verstehen, wenn Sie die zugrunde liegende Logik durchschaut haben. Die Zelle besitzt einen Kurzzeitstatusvektor und einen Langzeitstatusvektor. Bei jedem Zeitschritt werden die Eingaben und der vorherige Kurzzeitstatus an eine einfache RNN-Zelle und drei Gates übergeben: Das Forget Gate entscheidet, was aus dem Langzeitstatus zu entfernen ist, das Input Gate entscheidet, welche Teile der Ausgabe der einfachen RNN-Zelle zum Langzeitstatus hinzuzufügen sind, und das Output Gate entscheidet, welche Teile des Langzeitstatus bei diesem Zeitschritt ausgegeben werden sollten (nach dem Durchlauf durch die tanh-Aktivierungsfunktion). Der neue Kurzzeitstatus entspricht dann der Ausgabe der Zelle (siehe Abbildung 15-9).
7. Eine RNN-Schicht ist erst einmal sequenziell: Um die Ausgaben bei Zeitschritt t zu berechnen, muss sie erst die Ausgaben aller vorherigen Zeitschritte

berechnen. Das macht ein Parallelisieren unmöglich. Ein 1-D-Convolutional-Layer eignet sich hingegen sehr gut zur Parallelisierung, da er zwischen den Zeitschritten keinen Status aufbewahren muss. Mit anderen Worten – er hat kein Gedächtnis. Die Ausgabe kann bei jedem Zeitschritt allein anhand eines kleinen Fensters mit Werten aus den Eingaben berechnet werden, ohne alle vergangenen Werte kennen zu müssen. Und da ein 1-D-Convolutional-Layer nicht rekurrent ist, leidet er weniger unter instabilen Gradienten. Einer oder mehrere 1-D-Convolutional-Layers können in einem RNN nützlich sein, um die Eingaben effizient vorzuverarbeiten – zum Beispiel zum Verringern ihrer zeitlichen Auflösung (Downsampling), womit die RNN-Schichten langfristige Muster besser erkennen können. Tatsächlich ist es möglich, nur mit Convolutional Layers zu arbeiten, was zum Beispiel bei der WaveNet-Architektur geschieht.

8. Um Videos anhand ihres visuellen Inhalts zu klassifizieren, könnten Sie beispielsweise einen Frame pro Sekunde nehmen, dann jeden Frame durch das gleiche Convolutional Neural Network laufen lassen (zum Beispiel ein vortrainiertes Xception-Modell, eventuell eingefroren, wenn Ihr Datensatz nicht zu groß ist), die Ausgabesequenz aus dem CNN an ein Sequence-to-Vector-RNN übergeben und schließlich dessen Ausgabe durch eine Softmax-Schicht schicken, wodurch Sie die Wahrscheinlichkeiten für alle Kategorien erhalten. Zum Trainieren würden Sie als Kostenfunktion die Kreuzentropie verwenden. Wollen Sie auch die Audiodaten zur Klassifikation einsetzen, könnten Sie einen Stack mit Strided-1-D-Convolutional-Layers nutzen, um die zeitliche Auflösung von Tausenden Audioframes pro Sekunde auf nur einen pro Sekunde zu verringern (um zur Anzahl an Bildern pro Sekunde zu passen), und die Ausgabesequenz mit den Eingaben des Sequence-to-Vector-RNN (entlang der letzten Dimension) verbinden.

Die Lösungen zu den Übungsaufgaben 9 und 10 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 16: Verarbeitung natürlicher Sprache mit RNNs und Attention

1. Zustandslose RNNs können nur Muster erfassen, deren Länge kleiner oder gleich der Größe des Fensters ist, mit dem das RNN trainiert wird. Umgekehrt können zustandsbehaftete RNNs langfristigere Muster erfassen. Aber das Implementieren eines zustandsbehafteten RNN ist viel schwerer – insbesondere das korrekte Vorbereiten des Datensatzes. Zudem funktionieren zustandsbehaftete RNNs nicht immer besser – zum Teil, da aufeinanderfolgende Batches nicht unabhängig und gleichverteilt (Independent and Identically Distributed, IID) sind. Die Gradientenmethode ist für Nicht-IID-Datensätze nicht so gut geeignet.

2. Wenn Sie einen Satz Wort für Wort übersetzen, ist das Ergebnis in der Regel furchtbar. Beispielsweise bedeutet der französische Satz »Je vous en prie« übersetzt »Bitte schön«, aber wenn Sie ihn Wort für Wort übersetzen, erhalten Sie »Ich Sie im beten«. Es ist viel besser, zuerst den ganzen Satz zu lesen und ihn dann zu übersetzen. Ein einfaches Sequenz-zu-Sequenz-RNN würde unmittelbar nach Lesen des ersten Worts mit dem Übersetzen beginnen, wohingegen ein Encoder-Decoder-RNN zuerst den gesamten Satz liest und diesen erst dann übersetzt. Natürlich könnte man sich auch ein einfaches Sequenz-zu-Sequenz-RNN vorstellen, das einfach nur Stille ausgibt, wenn es sich beim nächsten Wort nicht sicher ist (wie es menschliche Dolmetscher bei einer Simultanübersetzung tun).
3. Eingabesequenzen variabler Länge können durch ein Auffüllen der kürzeren Sequenzen verarbeitet werden, sodass alle Sequenzen in einem Batch die gleiche Länge haben, und indem Maskierung verwendet wird, um sicherzustellen, dass das RNN das Padding-Token ignoriert. Für eine bessere Performance könnten Sie auch Batches erstellen, die Sequenzen ähnlicher Länge enthalten. Ragged-Tensoren können Sequenzen variabler Länge aufnehmen, und Keras unterstützt sie mittlerweile auch, was die Arbeit mit Eingabesequenzen variabler Länge deutlich vereinfachen wird (aktuell funktionieren Ragged-Tensoren allerdings noch nicht als Ziele auf der GPU). Wenn es um Ausgabesequenzen variabler Länge geht, müssen Sie bei Kenntnis der Länge im Voraus (zum Beispiel weil Sie wissen, dass sie genauso lang wie die Eingabesequenz ist) nur die Verlustfunktion so konfigurieren, dass sie Tokens nach dem Ende der Sequenz ignoriert. Ebenso sollte der Code, den das Modell verwendet, Tokens nach dem Ende der Sequenz ignorieren. Aber im Allgemeinen ist die Länge der Ausgabesequenz nicht im Voraus bekannt, daher liegt die Lösung darin, das Modell so zu trainieren, dass es am Ende jeder Sequenz ein End-of-Sequence-Token ausgibt.
4. Beam Search ist eine Technik, die zum Verbessern der Leistung eines trainierten Encoder-Decoder-Modells dient, zum Beispiel in einem Übersetzungssystem. Der Algorithmus merkt sich eine kurze Liste der k vielversprechendsten Ausgabesätze (zum Beispiel die Top 3), und bei jedem Decoder-Schritt versucht er, sie um ein Wort zu erweitern. Dann merkt er sich wieder nur die k wahrscheinlichsten Sätze. Der Parameter k wird als *Beam Width* bezeichnet – je größer er ist, desto mehr CPU und RAM wird gebraucht, aber desto genauer wird auch das System sein. Statt bei jedem Schritt gierig das wahrscheinlichste nächste Wort zu wählen, um einen einzelnen Satz zu erweitern, erlaubt diese Technik dem System, mehrere vielversprechende Sätze simultan zu untersuchen. Zudem kann sie gut parallelisiert werden. Sie können die Beam Search durch das Schreiben einer eigenen Speicherzelle implementieren. Alternativ stellt die seq2seq-API von TensorFlow Addons eine Implementierung zur Verfügung.

5. Ein Attention-Mechanismus ist eine Technik, die ursprünglich in Encoder-Decoder-Modellen genutzt wurde, um dem Decoder einen direkteren Zugriff auf die Eingabesequenzen zu erlauben und es ihm damit zu ermöglichen, mit längeren Sequenzen umzugehen. Bei jedem Decoder-Zeitschritt werden der aktuelle Status des Decoders und die vollständige Ausgabe des Encoders von einem Alignment-Modell verarbeitet, das einen Alignment Score für jeden Eingabezeitschritt ausgibt. Dieser Score gibt an, welcher Teil der Eingabe für den aktuellen Decoder-Zeitschritt am relevantesten ist. Die gewichtete Summe der Encoder-Ausgabe (gewichtet nach dem Alignment Score) wird dann an den Decoder übergeben, der den nächsten Decoder-Status und die Ausgabe für diesen Zeitschritt erzeugt. Der Hauptvorteil des Attention-Mechanismus ist, dass das Encoder-Decoder-Modell erfolgreich längere Eingabesequenzen verarbeiten kann. Ein weiterer Vorteil ist, dass das Modell durch die Alignment Scores leichter zu debuggen und zu interpretieren sein wird: Macht das Modell beispielsweise einen Fehler, können Sie sich anschauen, auf welchen Teil der Eingabe es seine Aufmerksamkeit gerichtet hat, was bei der Diagnose des Problems helfen kann. Ein Attention-Mechanismus ist zudem in den Multi-Head-Attention-Schichten das Herz der Transformer-Architektur (siehe nächste Antwort).
6. Die wichtigste Schicht in der Transformer-Architektur ist die Multi-Head-Attention-Schicht (die ursprüngliche Transformer-Architektur enthielt 18 davon, unter anderem 6 Masked Multi-Head-Attention-Schichten). Sie bildet auch den Kern von Sprachmodellen wie BERT oder GPT-2. Mit dieser Schicht kann das Modell herausfinden, welche Wörter miteinander am meisten verbunden sind, und dann die Repräsentation jedes Worts mit diesen kontextuellen Hinweisen verbessern.
7. Sampled Softmax wird beim Trainieren eines Klassifikationsmodells verwendet, wenn es viele Kategorien gibt (zum Beispiel Tausende). Er berechnet eine Näherung des Kreuzentropieverlusts, die auf dem vom Modell für die korrekte Kategorie vorhergesagten Logit und den vorhergesagten Logits für ein Sample falscher Wörter basiert. Das beschleunigt das Training im Vergleich zum Berechnen des Softmax über alle Logits und dem darauffolgenden Schätzen des Kreuzentropieverlusts deutlich. Nach dem Training kann das Modell normal mit der regulären Softmax-Funktion verwendet werden, um alle Kategorienwahrscheinlichkeiten basierend auf allen Logits zu berechnen.

Die Lösungen zu den Übungsaufgaben 8 bis 11 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 17: Autoencoder, GANs und Diffusionsmodelle

1. Einige der wichtigsten Aufgaben, für die Autoencoder verwendet werden, sind folgende:
   - Extrahieren von Merkmalen
   - Unüberwachtes Vortrainieren
   - Dimensionsreduktion
   - Generative Modelle
   - Erkennen von Anomalien (ein Autoencoder ist grundsätzlich schlecht im Rekonstruieren von Ausreißern)
2. Wenn Sie einen Klassifikator trainieren möchten und reichlich ungelabelte Trainingsdaten besitzen, aber nur wenige Tausend gelabelte Datenpunkte, können Sie zuerst einen Deep Autoencoder auf dem gesamten Datensatz trainieren (gelabelt und ungelabelt), anschließend dessen untere Hälfte für den Klassifikator verwenden (d.h. die Schichten bis einschließlich der Schicht mit den Codings wiederverwenden) und den Klassifikator mit den gelabelten Daten trainieren. Wenn Sie wenige gelabelte Daten haben, sollten Sie die wiederverwendeten Schichten beim Trainieren des Klassifikators einfrieren.
3. Dass ein Autoencoder die Eingabedaten perfekt rekonstruiert, bedeutet nicht unbedingt, dass es ein guter Autoencoder ist; es könnte auch einfach ein übervollständiger Autoencoder sein, der gelernt hat, die Eingaben in die Codings und von dort in die Ausgabe zu überführen. Selbst wenn die Schicht mit den Codings nur aus einem einzelnen Neuron besteht, könnte ein sehr tiefer Autoencoder lernen, jeden Trainingsdatenpunkt auf ein anderes Coding zu übertragen (z.B. könnte der erste Datenpunkt als 0,001 codiert sein, der zweite als 0,002, der dritte als 0,003 und so weiter). So könnte er lernen, den richtigen Trainingsdatenpunkt »auswendig« zu rekonstruieren. Damit würden die Eingabedaten perfekt rekonstruiert, ohne dass irgendein nützliches Muster in den Daten erlernt würde. In der Praxis ist ein derartiges Mapping unwahrscheinlich, es hebt aber den Umstand hervor, dass perfekte Rekonstruktionen keine Garantie dafür sind, dass der Autoencoder etwas Nützliches gelernt hat. Wenn die Rekonstruktionen dagegen sehr schlecht sind, können Sie sich sicher sein, dass der ganze Autoencoder nichts taugt. Um die Leistung eines Autoencoder zu evaluieren, können Sie den Verlust bei der Rekonstruktion bestimmen (z.B. den MSE, die mittlere quadrierte Differenz zwischen Ausgaben und Eingaben). Auch hier ist ein hoher Verlustbetrag bei der Rekonstruktion ein sicheres Zeichen für einen schlechten Autoencoder, aber ein niedriger Betrag ist keine Garantie für einen guten. Sie sollten den Autoencoder auch entsprechend seiner Anwendung evaluieren. Wenn Sie ihn beispielsweise zum

unüberwachten Vortrainieren einsetzen möchten, sollten Sie auch die Leistung des Klassifikators auswerten.

4. Ein untervollständiger Autoencoder ist einer, dessen Coding-Schicht kleiner als die Ein- und Ausgabeschichten ist. Ist sie größer, handelt es sich um einen übervollständigen Autoencoder. Die Gefahr bei einem stark untervollständigen Autoencoder ist, dass er an der Rekonstruktion der Eingaben scheitert. Die Gefahr bei einem übervollständigen Autoencoder ist, dass er die Eingaben einfach in die Ausgaben kopiert, ohne irgendwelche nützlichen Merkmale zu erlernen.
5. Sie können die Gewichte einer Encoder-Schicht mit der entsprechenden Decoder-Schicht koppeln, indem Sie die Gewichte des Decoders den transponierten Gewichten des Encoders gleichsetzen. Damit sinkt die Anzahl der Modellparameter auf die Hälfte, das Training konvergiert schneller und mit weniger Trainingsdaten, und das Risiko für Overfitting sinkt.
6. Ein generatives Modell ist ein Modell, das zufällig Ausgaben generiert, die den Trainingsdatenpunkten ähnlich sind. Beispielsweise könnte ein erfolgreich auf dem MNIST-Datensatz trainiertes generatives Modell dazu verwendet werden, zufällig realistische Bilder von Ziffern zu erzeugen. Die Ausgaben sind normalerweise ähnlich wie die Trainingsdaten verteilt. Da MNIST viele Bilder jeder Ziffer enthält, würde das generative Modell etwa die gleiche Anzahl Bilder für jede Ziffer liefern. Einige generative Modelle lassen sich parametrisieren – um beispielsweise nur bestimmte Ausgaben zu erzeugen. Variational Autoencoder sind ein Beispiel für einen generativen Autoencoder.
7. Ein Generative Adversarial Network ist eine neuronale Netzarchitektur aus zwei Teilen – dem Generator und dem Diskriminator, die gegensätzliche Ziele verfolgen. Ziel des Generators ist, Instanzen zu erzeugen, die denen im Trainingsdatensatz ähneln, um den Diskriminator zu täuschen. Der Diskriminator muss die echten Instanzen von den generierten Instanzen unterscheiden. Bei jeder Trainingsiteration wird der Diskriminator wie ein normaler Binärklassifikator trainiert, dann wird der Generator darauf trainiert, den Fehler des Diskriminators zu maximieren. GANs werden für fortgeschrittene Bildverarbeitungsaufgaben genutzt, wie zum Beispiel zum Verbessern der Auflösung, zum Einfärben, zur Bildbearbeitung (Objekte durch realistische Hintergründe ersetzen), zum Umwandeln einer einfachen Skizze in ein fotorealistisches Bild oder zum Vorhersagen der nächsten Frames in einem Video. Außerdem kommen sie beim Augmentieren eines Datensatzes zum Einsatz (um andere Modelle zu trainieren), beim Erzeugen anderer Arten von Daten (zum Beispiel Text, Audio oder Zeitserien) und beim Auffinden und Beheben von Schwächen in anderen Modellen.
8. Das Trainieren eines GAN ist aufgrund der komplexen Dynamiken zwischen dem Generator und dem Diskriminator immer schwierig. Das größte Problem ist der Mode Collapse, bei dem der Generator Ausgaben mit sehr wenig Diver-

sität erzeugt. Zudem kann das Training furchtbar instabil sein: Es kann gut losgehen und dann plötzlich ohne offensichtlichen Grund oszillieren oder divergieren. GANs reagieren auch sehr empfindlich auf die Wahl der Hyperparameter.

9. Diffusionsmodelle sind gut darin, unterschiedliche und qualitativ hochwertige Bilder zu generieren. Sie lassen sich auch viel einfacher trainieren als GANs. Aber im Vergleich zu GANs und VAEs sind sie beim Generieren viel langsamer, da sie jeden Schritt im Rückwärts-Diffusionsprozess durchlaufen müssen.

Die Lösungen zu den Übungsaufgaben 10 bis 13 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 18: Reinforcement Learning

1. Reinforcement Learning ist ein Teilgebiet des Machine Learning, bei dem Agenten Aktionen in einer Umwelt so auszuführen lernen, dass sie über einen längeren Zeitraum maximale Belohnungen erzielen. Es gibt viele Unterschiede zwischen RL und gewöhnlichem überwachtem und unüberwachtem Lernen. Hier sind einige davon:
   - Beim überwachten und unüberwachten Lernen ist das Ziel, Muster in den Daten zu entdecken und sie zur Vorhersage zu nutzen. Beim Reinforcement Learning geht es darum, eine gute Policy zu finden.
   - Im Gegensatz zum überwachten Lernen bekommt der Agent keine expliziten »richtigen« Antworten gezeigt. Er muss diese durch Versuch und Irrtum herausfinden.
   - Im Gegensatz zum unüberwachten Lernen gibt es eine Art Überwachung in Form von Belohnungen. Wir sagen dem Agenten nicht, wie er seine Aufgabe ausführen soll, aber wir verraten ihm, wann er Fortschritte erzielt oder scheitert.
   - Beim Reinforcement Learning muss der Agent die richtige Balance zwischen dem Erkunden seiner Umwelt zum Entdecken neuer Belohnungsmöglichkeiten und dem Nutzen bereits bekannter Belohnungsquellen finden. Im Gegensatz dazu kümmern sich überwachte und unüberwachte Lernsysteme nicht um die Erkundung; sie verwenden einfach nur die erhaltenen Trainingsdaten.
   - Beim überwachten und unüberwachten Lernen sind die Trainingsdatenpunkte normalerweise unabhängig voneinander (meistens sind sie durchmischt). Beim Reinforcement Learning sind aufeinanderfolgende Beobachtungen grundsätzlich *nicht* voneinander unabhängig. Ein Agent kann eine Weile im gleichen Gebiet einer Umwelt verweilen, bevor er weiterzieht, sodass aufeinanderfolgende Beobachtungen stark miteinander kor-

relieren. In einigen Fällen wird ein Replay Buffer (Speicher) verwendet, um sicherzustellen, dass der Trainingsalgorithmus halbwegs unabhängige Beobachtungen erhält.

2. Hier sind, zusätzlich zu den in Kapitel 18 erwähnten, einige mögliche Anwendungen für Reinforcement Learning:

   Personalisierte Musik
   : Die Umwelt ist das personalisierte Webradio des Nutzers. Der Agent ist eine Software, die entscheidet, welcher Song dem Nutzer als Nächstes vorgespielt wird. Die möglichen Aktionen sind, einen Song aus dem Verzeichnis abzuspielen (und dabei einen Song auszuwählen, der dem Nutzer gefällt) oder Werbung abzuspielen (und eine Werbung auszuwählen, für die sich der Nutzer interessiert). Der Agent erhält jedes Mal eine kleine Belohnung, wenn der Nutzer sich einen Song anhört, und eine größere, wenn er einen Werbespot verfolgt. Er erhält eine negative Belohnung, wenn der Nutzer einen Song oder Werbespot überspringt, und eine stark negative, wenn er abschaltet.

   Marketing
   : Die Umwelt ist die Marketingabteilung Ihres Unternehmens. Der Agent ist die Software, die anhand von Profilen und früheren Kaufentscheidungen von Kunden entscheidet, an welche Kunden eine Mailkampagne gerichtet wird. Der Agent erhält eine negative Belohnung für die Kosten der Mailkampagne und eine positive Belohnung für den geschätzten, durch diese Kampagne generierten Gewinn.

   Auslieferung von Produkten
   : Der Agent kontrolliert eine Flotte von Lieferfahrzeugen und entscheidet, was diese bei den Depots aufnehmen, wohin sie fahren, wo sie ausladen und so weiter. Er erhält positive Belohnungen für jedes pünktlich ausgelieferte Produkt und negative Belohnungen für eine verspätete Auslieferung.

3. Beim Abschätzen des Werts einer Aktion versuchen Reinforcement-Learning-Algorithmen normalerweise, alle durch diese Aktion erzeugten Belohnungen aufzusummieren. Dabei erhalten unmittelbare Belohnungen ein höheres Gewicht und spätere ein geringeres (berücksichtigt wird, dass Aktionen die nahe Zukunft stärker beeinflussen als die ferne). Um dies zu modellieren, wird normalerweise bei jedem Schritt ein Discountfaktor mit eingerechnet. Beispielsweise würde bei einem Discountfaktor von 0,9 eine zwei Schritte in der Zukunft liegende Belohnung der Höhe 100 nur mit $0{,}9^2 \times 100 = 81$ beim Wert der entsprechenden Aktion berücksichtigt. Sie können sich den Discountfaktor als ein Maß dafür vorstellen, wie stark die Zukunft im Vergleich zur Gegenwart gewertet wird: Liegt er sehr nah an 1, ist die Zukunft beinahe genauso viel wert wie die Gegenwart. Liegt er nahe bei 0, zählen nur unmittelbare Belohnungen. Natürlich hat dies einen immensen Einfluss auf die optimale Policy: Wenn Sie der Zukunft einen Wert beimessen, sind Sie vielleicht

bereit, eine Menge unmittelbarer Schmerzen für die Aussicht auf eventuelle Gewinne auf sich zu nehmen. Messen Sie der Zukunft dagegen keinen Wert bei, schnappen Sie sich einfach nur jede unmittelbare Belohnung, an der Sie vorbeikommen, und investieren nie in die Zukunft.

4. Um die Leistung eines Agenten beim Reinforcement Learning zu messen, können Sie einfach dessen Belohnungen aufsummieren. In einer simulierten Umgebung können Sie viele Episoden ausführen und sich die durchschnittliche Summe der Belohnungen ansehen (ebenso Minimum, Maximum, Standardabweichung und so weiter).
5. Das Problem bei der Kreditzuweisung ist der Umstand, dass ein Agent beim Reinforcement Learning beim Erhalten einer Belohnung nicht direkt erfährt, welche seiner vorausgegangenen Aktionen zu dieser Belohnung beigetragen haben. Dies ist normalerweise der Fall, wenn zwischen der Aktion und der sich ergebenden Belohnung ein langer Zeitraum lag (z. B. können beim Atari-Spiel *Pong* zwischen dem Schlagen des Balls durch den Agenten und dem Erzielen eines Punkts mehrere Dutzend Zeitschritte liegen). Das Problem lässt sich verringern, indem man dem Agenten, wenn möglich, kurzfristigere Belohnungen bereitstellt. Dies erfordert Hintergrundwissen zur Aufgabe. Wenn wir beispielsweise einem Agenten das Schachspiel beibringen möchten, könnten wir ihm jedes Mal eine Belohnung geben, wenn er eine gegnerische Figur schlägt, anstatt nur Belohnungen für gewonnene Partien zu verteilen.
6. Ein Agent verbringt häufig eine Weile in der gleichen Region seiner Umwelt. Daher sind dessen Erfahrungen in diesem Zeitraum einander sehr ähnlich. Dies kann zu einem Bias im Lernalgorithmus führen. Die Policy kann dann auf diese Region der Umwelt optimiert sein, aber nicht so gut außerhalb funktionieren. Um dieses Problem zu beheben, können Sie einen Replay Buffer nutzen. Anstatt nur die jüngsten Erfahrungen zum Lernen heranzuziehen, verwendet der Agent zum Lernen eine Sammlung von Erfahrungen aus seiner jüngeren und früheren Vergangenheit. (Vielleicht träumen wir deshalb: um unsere Erfahrungen des Tags erneut abzuspielen und besser aus ihnen zu lernen?)
7. Ein Off-Policy-RL-Algorithmus erlernt den Wert der optimalen Policy (d. h. die bei optimalem Verhalten des Agenten zu erwartende Summe der Belohnungen unter Berücksichtigung der Discount-Rate), während der Agent eine andere Policy verfolgt. Q-Learning ist ein gutes Beispiel für solch einen Algorithmus. Im Gegensatz dazu erlernt ein On-Policy-Algorithmus den Wert der vom Agenten tatsächlich ausgeführten Policy, was die Erkundung und Nutzung einschließt.

Die Lösungen zu den Übungsaufgaben 8, 9 und 10 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.

Kapitel 19: TensorFlow-Modelle skalierbar trainieren und deployen

1. Ein SavedModel enthält ein TensorFlow-Modell mit seiner Architektur (einem Rechengraphen) und den Gewichten. Es wird als Verzeichnis mit einer Datei *saved_model.pb* abgelegt, in der der Rechengraph definiert ist (als serialisierter Protocol Buffer), und einem Unterverzeichnis *variables* mit den Variablenwerten. Bei Modellen mit einer großen Zahl von Gewichten können diese Werte auf mehrere Dateien verteilt sein. Ein SavedModel enthält zudem ein Unterverzeichnis *assets*, in dem sich zusätzliche Daten befinden können, wie zum Beispiel Vokabulardateien, Klassennamen oder Beispielinstanzen für dieses Modell. Wenn man genau sein will, kann ein SavedModel auch noch einen oder mehrere *Metagraphen* besitzen. Dabei handelt es sich um einen Rechengraphen zusammen mit Definitionen von Funktionssignaturen (einschließlich der Eingabe- und Ausgabenamen, -typen und -formen). Jeder Metagraph wird durch einen Satz Tags identifiziert. Um ein SavedModel zu untersuchen, können Sie das Befehlszeilentool `saved_model_cli` nutzen oder es einfach mit `tf.saved_model.load()` laden und in Python unter die Lupe nehmen.
2. TF Serving ermöglicht Ihnen, mehrere TensorFlow-Modelle (oder mehrere Versionen des gleichen Modells) zu deployen und über eine REST-API oder eine gRPC-API ganz einfach all Ihren Anwendungen bereitzustellen. Wenn Sie Ihre Modelle in Ihren Anwendungen direkt verwenden, wird es schwerer, eine neue Version eines Modells in allen Anwendungen auszurollen. Es würde mehr Aufwand bedeuten, einen eigenen Microservice zu implementieren, der ein TF-Modell verpackt, und es wäre schwer, mit den Features von TF Serving mitzuhalten: Es kann ein Verzeichnis überwachen und automatisch die Modelle deployen, die dort abgelegt werden, und Sie müssen keine Ihrer Anwendungen ändern oder auch nur neu starten, um von den neuen Modellversionen zu profitieren. Es ist schnell, umfassend getestet und skaliert sehr gut. Es unterstützt A/B-Testing für experimentelle Modelle und das Deployen einer neuen Modellversion an nur eine Untergruppe der Anwender (das nennt sich dann ein *Canary*). TF Serving kann zudem einzelne Requests in Batches zusammenfassen, um sie gemeinsam auf der GPU laufen zu lassen. Um TF Serving zu deployen, können Sie es aus seinem Quellcode installieren, aber es ist viel einfacher, es über ein Docker-Image zu installieren. Um ein Cluster mit TF-Serving-Docker-Images zu deployen, können Sie ein Orchestrierungstool wie Kubernetes verwenden oder auf eine vollständig gehostete Lösung wie Google Vertex AI zurückgreifen.
3. Um ein Modell auf mehreren Instanzen von TF Serving zu deployen, müssen Sie diese Instanzen nur so konfigurieren, dass sie alle das gleiche *models*-Verzeichnis überwachen, und dann Ihr neues Modell als SavedModel in einem Unterverzeichnis ablegen.

4. Die gRPC-API ist effizienter als die REST-API. Aber deren Clientbibliotheken stehen nicht so umfassend zur Verfügung, und wenn Sie beim Einsatz der REST-API die Komprimierung einschalten, können Sie fast die gleiche Performance erhalten. Daher ist die gRPC-API dann sinnvoll, wenn Sie die höchstmögliche Leistung brauchen und die Clients nicht auf die REST-API beschränkt sind.
5. Um die Größe eines Modells zu reduzieren, damit es auf einem mobilen oder Embedded Device laufen kann, nutzt TFLite eine Reihe von Techniken:
   - Es stellt einen Konverter bereit, der ein SavedModel optimieren kann: Er dampft das Modell ein und verringert seine Latenz. Dazu schneidet er alle Operationen weg, die zum Treffen von Vorhersagen nicht notwendig sind (wie zum Beispiel Trainingsoperationen), und optimiert und verschmilzt wann immer möglich Operationen.
   - Der Konverter kann zudem eine Post-Training-Quantisierung durchführen: Diese Technik verringert die Modellgröße drastisch, sodass sich das Modell dann schneller herunterladen und abspeichern lässt.
   - Es sichert das optimierte Modell im FlatBuffer-Format, das ohne Parsen direkt ins RAM geladen werden kann. Damit verringern sich Ladezeiten und Speicherbedarf.
6. Quantisierungsbewusstes Training fügt beim Training künstliche Quantisierungsoperationen hinzu. Damit kann das Modell lernen, das Quantisierungsrauschen zu ignorieren – die finalen Gewichte werden dann robuster darauf reagieren.
7. Für parallelisierte Modelle zerschneiden Sie Ihr Modell in mehrere Teile und führen diese parallel auf mehreren Devices aus, womit Sie das Modell beim Training oder bei der Inferenz hoffentlich beschleunigen. Bei parallelisierten Daten erstellen Sie mehrere gleiche Repliken Ihres Modells und deployen sie auf mehrere Devices. Während des Trainings erhält jede Replik bei jeder Iteration einen anderen Datenbatch, und sie berechnet die Gradienten des Verlusts bezüglich der Modellparameter. Bei synchron parallelisierten Daten werden dann die Gradienten aller Repliken zusammengeführt, und der Optimierer führt einen Gradientenschritt durch. Die Parameter können zentral (zum Beispiel auf Parameterservern) oder verteilt über alle Repliken vorliegen und mithilfe von AllReduce synchron gehalten werden. Bei asynchron parallelisierten Daten liegen die Parameter zentral vor, und die Repliken laufen unabhängig voneinander. Jede Replik aktualisiert die zentralen Parameter direkt am Ende jeder Trainingsiteration, ohne auf die anderen Repliken warten zu müssen. Um das Training zu beschleunigen, sind parallelisierte Daten im Allgemeinen besser als parallelisierte Modelle. Das liegt vor allem daran, dass zwischen den Devices weniger Kommunikation erforderlich ist. Zudem lässt sie sich viel leichter implementieren und funktioniert für jedes Modell gleich, während Sie bei parallelisierten Modellen das Modell analysieren müssen, um herauszufin-

den, wie Sie es am besten unterteilen können. Aktuell gibt es immer wieder neue Erkenntnisse in diesem Bereich (aus denen zum Beispiel PipeDream oder Pathways erwachsen sind), daher wird vermutlich eine Mischung aus Modell-Parallelisierung und Daten-Parallelisierung der zukünftige Weg sein.

8. Beim Trainieren eines Modells auf mehreren Servern können Sie die folgenden Verteilstrategien anwenden:
   - Die `MultiWorkerMirroredStrategy` setzt auf gespiegelte parallelisierte Daten. Das Modell wird auf alle verfügbaren Server und Devices deployt, jede Replik erhält bei jeder Trainingsiteration einen anderen Batch mit Daten und berechnet ihre eigenen Gradienten. Dann wird mit einer verteilten AllReduce-Implementierung (standardmäßig NCCL) der Mittelwert der Gradienten bestimmt und auf alle Repliken verteilt, wonach dort überall der gleiche Gradientenschritt ausgeführt wird. Diese Strategie lässt sich am einfachsten einsetzen, da alle Server und Devices genau gleich behandelt werden und sie ziemlich gut funktioniert. Im Allgemeinen sollten Sie diese Strategie verfolgen. Ihre größte Einschränkung ist, dass das Modell auf jeder Replik ins RAM passen muss.
   - Die `ParameterServerStrategy` nutzt asynchron parallelisierte Daten. Das Modell wird auf alle Devices auf allen Workern deployt, und die Parameter werden per Sharding über alle Parameterserver verteilt. Jeder Worker besitzt seine eigene Trainingsschleife und läuft asynchron zu den anderen Workern. Bei jeder Trainingsiteration erhält jeder Worker seinen eigenen Datenbatch und holt sich vom Parameterserver die neueste Version der Modellparameter. Dann berechnet er die Gradienten des Verlusts bezüglich dieser Parameter und schickt sie an die Parameterserver. Schließlich führen die Parameterserver mit diesen Gradienten einen Gradientenschritt durch. Diese Strategie ist meist langsamer als die vorherige Strategie, und sie lässt sich auch nur aufwendiger deployen, da dafür Parameterserver verwaltet werden müssen. Aber sie kann in manchen Situationen nützlich sein, insbesondere wenn die asynchronen Updates Vorteile bringen – beispielsweise zum Verringern von I/O-Flaschenhälsen. Das hängt von vielen Faktoren ab, unter anderem von der Hardware, der Netzwerktopologie, der Anzahl an Servern und der Modellgröße – es kann bei Ihnen also ganz anders aussehen.

Die Lösungen zu den Übungsaufgaben 9, 10 und 11 finden Sie in den Jupyter Notebooks unter *https://homl.info/colab3*.