Woche 11 Programmierung: Tensorboard

Skript

Erarbeitet von
Ann-Kathrin Selker

Lernziele

- die Anwendung Tensorboard kennen lernen
- Sinn und Anwendung von Callbacks erklären können

Inhalt

Während des Trainings gibt es viele Parameter, die die Qualität des Trainings beeinflussen. Häufig braucht es viele Trainingsdurchläufe, bis die richtige Kombination an Parametern gefunden wird. Doch wie kannst du schnell den Trainingserfolg der verschiedenen Parameterkombinationen vergleichen?

Projekt „KI für alle"
HeiCAD – Heine Center for Artificial Intelligence and Data Science

Einstieg

Ein einfaches Mittel, um die gewählten Parameter und Trainingserfolge zu dokumentieren, sind log-Dateien, auch Protokolldatei genannt. Log-Dateien werden automatisch durch ein Computersystem erstellt. So werden vielleicht deine Arbeitszeiten protokolliert, ein Flugschreiber zeichnet alle Flugdaten und -parameter auf, und dein Computer erstellt Berichte, wenn ein Programm abstürzt. Das Modul Tensorflow bietet nicht nur die Möglichkeit, Log-Dateien während des Trainings zu erstellen, sondern beinhaltet auch eine Anwendung namens Tensorboard, die dir unter anderem diese Log-Dateien mithilfe eines Dashboards visualisiert. In diesem Video zeige ich dir, wie du dir mithilfe von tensorboard den Optimierungsprozess erleichtern kannst.

Modellerstellung

Die Optionen, die dir Tensorboard bietet, sind gerade am Anfang noch viel zu ausführlich und sicherlich etwas überwältigend. Trotzdem ist diese Anwendung ein guter Startpunkt, wenn du dein Modell optimieren möchtest. Du kannst beispielsweise bereits Unter- und Überanpassung feststellen. Damit du Tensorboard in deinem Jupyter Notebook verwenden kannst, musst du die Tensorboard-Erweiterung mit diesem Befehl laden. Das Prozentzeichen ist kein Tippfehler, sondern gehört zu dem Befehl dazu.

```
%load_ext tensorboard
```

Außerdem importieren wir das Modul tensorflow. Als Beispiel betrachten wir den Datensatz MNIST, in dem Graustufenbilder von handgeschriebenen Ziffern bereitgestellt werden, z.B. diese hier.

Einblendung MNIST-Bilder (Quelle [1])

MNIST ist bereits in dem Modul Keras enthalten und kann mit diesem Befehl geladen werden.

```
(x_train, y_train),(x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

Ziel ist es, jedem Eingabebild eine der zehn Klassen 0 bis 9 zuzuordnen. Wir normalisieren unsere Bilder, indem wir alle Pixelwerte durch 255 teilen, und zweigen von den 60 000 Trainingsbildern noch die hinteren 10 000 Bilder als Validierungsdaten ab, da der MNIST-Datensatz keine eigenen Validierungsdaten zur Verfügung stellt.

```
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train, x_val = x_train[:50000], x_train[50000:]
y_train, y_val = y_train[:50000], y_train[50000:]
```

Das neuronale Netz definieren wir dann folgendermaßen:

Wir starten mit einem neuronalen Netz mit einer Eingabeschicht, einer versteckten Schicht mit 128 Neuronen, und mit der Ausgabeschicht. Bei den MNIST-Bildern handelt es sich um 28 x 28 Pixel-Bilder, also ist unsere Eingabe auch eine Matrix mit der Dimension 28 x 28. Wir haben zehn verschiedene Klassen, daher benötigt unsere Ausgabeschicht auch zehn Neuronen. Die Aktivierungsfunktion softmax gibt einen Vektor zurück, der für jede Klasse eine Wahrscheinlichkeit angibt, dass es sich bei der Eingabe um ein Element dieser Klasse handelt. Teilt die softmax-Funktion also z.B. der Klasse 4 die Wahrscheinlichkeit 0,93, also 93% zu, dann handelt es sich laut dem Modell mit 93%-iger Wahrscheinlichkeit um ein Bild von der Ziffer 4. Da es sich bei den Vektoreinträgen um Wahrscheinlichkeiten handelt, müssen sich alle Einträge zu 1 (also 100%) aufaddieren. Die Klasse mit der höchsten Wahrscheinlichkeit ist dann die Vorhersage des Modells. Die versteckte Schicht benutzt die Standardfunktion "relu" als Aktivierungsfunktion. Es gibt bei der Erstellung der Schichten die Möglichkeit, diesen Namen zu geben. Wenn du diese Möglichkeit nutzt, kannst du dir später bei Tensorboard die Performance jeder dieser Schichten anzeigen lassen.

```
model = tf.keras.models.Sequential([
   tf.keras.layers.Flatten(input_shape=(28, 28), name='input'),
   tf.keras.layers.Dense(128, activation='relu', name='dense1'),
   tf.keras.layers.Dense(10, activation='softmax', name='output')
 ])
```

Als Einsteiger ist es noch nicht so wichtig, die mathematischen Hintergründe der Verlust- und Optimierungsfunktionen zu kennen. Es reicht, sich darüber schlau zu machen, welche Funktionen bei welchen Zielsetzungen verwendet werden. Bedenke aber immer, dass die Wahl von Verlust- und Optimierungsfunktion einen großen Einfluss auf die Qualität und Dauer des Trainings haben! Zur Kompilierung benutzen wir in diesem Video diese Einstellungen.
Zuerst muss mit optimizer angegeben werden, mit welchem Optimierungsalgorithmus gearbeitet werden soll. Zur Erinnerung: Der Optimierungsalgorithmus minimiert die Verlustfunktion. Hier benutzen wir den Optimierungsalgorithmus 'Adam', der auf dem stochastischen Gradientenabstieg beruht. Die voreingestellte Schrittweite beim Gradientenabstieg (im englischen learning rate genannt) beträgt 0.001. Diese kannst du aber natürlich selber anpassen.
Da unser Ziel eine Kategorisierung ist, eignet sich die Verlustfunktion Kreuzentropie gut. Die Kreuzentropie gibt es in Tensorflow in mehreren Varianten: Wenn du nur zwei Klassen hast, solltest du die binäre Kreuzentropie (binary_crossentropy) benutzen, ansonsten die kategorische Kreuzentropie (categorical_crossentropy). Bei der kategorischen Variante muss dein Label als Vektor angegeben sein, bei dem eine 1 an der Stelle der entsprechenden Klasse und eine 0 sonst steht. Ein Beispiel dafür ist dieser Vektor [0 0 0 0 0 0 0 1 0 0], der bei MNIST für die Klasse 7 steht. Ist dein Label hingegen als ganze Zahl gegeben, also z.B. 7, musst du die Funktion sparse_categorical_crossentropy benutzen. Das Ergebnis beider Funktionen ist aber identisch. Wenn wir uns das Label eines beliebigen Datenobjekts ausgeben lassen, sehen wir, dass wir es hier mit ganzen Zahlen als Label zu tun haben, daher benutzen wir die Verlustfunktion sparse_categorical_crossentropy.

```
y_train[123]
```

Als letztes müssen wir noch angeben, an welcher Metrik wir interessiert sind. Wie gewohnt verwenden wir hier die Genauigkeit, also Accuracy. Achtung: Wir müssen eine Liste übergeben, daher achte auf die eckigen Klammern! Wir sind hier zwar nur an einer Metrik interessiert, aber es ist auch möglich, mehrere zu übergeben. Die Metriken werden beim Training nicht verwendet und dienen nur dazu, dir als Benutzer den Erfolg des Trainings zu. veranschaulichen.

```
model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
```

Callbacks

Als nächstes erstellen wir einen sogenannten Callback. Mithilfe von Callbacks kann das Verhalten beim Trainieren angepasst werden. Es handelt sich dabei um eine Funktion, die beim Training mit übergeben und an gewissen Stellen im Training aufgerufen wird, zum Beispiel vor jeder neuen Epoche. Zur Erinnerung: Eine Epoche ist ein Durchlauf durch deine Trainingsdaten. Callbacks können benutzt werden, um z.B. mithilfe von Logdateien das Training zu dokumentieren oder das Training beim Erreichen bestimmter Kriterien frühzeitig abzubrechen. Ein beliebtes Kriterium hier ist z.B. das Erreichen einer bestimmten Schranke bei den Metriken. Wenn du für deine Anwendung nur eine Genauigkeit von 95% benötigst, kannst du das Training abbrechen, sobald nach dem Durchlaufen einer Epoche eine Genauigkeit von mindestens 95% erreicht wird. Du kannst deinen eigenen Callback schreiben oder vordefinierte verwenden. Für dieses Video benutzen wir den Tensorboard-Callback. Hier geben wir als Parameter den Pfad zu einem Log-Ordner an. Dabei steht logdir für das englische log directory, übersetzt als Log-Ordner. Wir benutzen zuerst "logs/fit/standard".

```
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="logs/fit/standard")
```

Jetzt sind wir endlich bereit zum Trainieren. Unsere Daten haben wir bereits in Trainingsdaten, Validierungsdaten und Testdaten aufgeteilt. Wir übergeben unsere Trainingsdaten, die dazugehörigen Label und unsere Validierungsdaten. Für den Anfang trainieren wir für acht Epochen. Zu guter Letzt muss noch der Tensorboard-Callback als Parameter beim Trainieren mit übergeben werden, damit er beim Training auch benutzt wird. Die eckigen Klammern sind nötig, da es sich hier um eine Liste handelt. Es können also auch mehrere Callbacks mit übergeben werden.

```
model.fit(x=x_train,
      y=y_train,
      epochs=8,
      validation_data=(x_val, y_val),
      callbacks=[tensorboard_callback])
```

Das Modell ist fertig trainiert, unsere Log-Daten sind erstellt, als nächstes öffnen wir die Tensorboard-Anwendung, um uns den Trainingserfolg grafisch anzeigen zu lassen. Im Jupyter-Notebook musst du dafür diesen Befehl verwenden. Mit dem %tensorboard öffnest du die Anwendung, und --logdir gibt an, dass Tensorboard den Log-Ordner, der dahinter steht, als Quelle für Log-Dateien verwenden soll. Möchtest du außerhalb von Jupyter Notebooks oder ähnlichen Notebooks die Anwendung Tensorboard verwenden, kannst du sie über die Konsole starten.

%tensorboard --logdir logs

Als nächstes siehst du, dass sich Tensorboard öffnet. Wir betrachten in diesem Video nur den Reiter Time Series. In der Mitte siehst du zuerst zwei Grafiken, einmal für unsere Metrik Genauigkeit, also Accuracy, einmal für den entstandenen Fehler. Du siehst, dass schon nach acht Epochen die Genauigkeit mit über 99% bei den Trainingsdaten extrem hoch ist. Auch die Werte für die Validierungsdaten sehen gut aus. Die Diagramme sind interaktiv, das heißt du kannst die genauen Werte sehen, wenn du mit der Maus über die Linien fährst. Wenn du nicht alle vorhandenen Logs angezeigt bekommen möchtest, kannst du links in der Anwendung die entsprechenden logs abwählen. Außerdem ist es hier möglich, die Farbe der Linien zu wechseln. Bei den Diagrammen selber gibt es die Möglichkeit, die dargestellten Datenobjekte herunterzuladen. Auf der rechten Seite gibt es einige Einstellungen bezüglich Skalierungen und Darstellung der Diagramme.

Einblendung Tensorboard

Dauer des Trainings

Als nächstes versuchen wir, künstlich Unteranpassung zu erzeugen, um die Graphen vergleichen zu können. Dafür nehmen wir eine extrem kleine Zahl als Neuronen in der versteckten Schicht, zum Beispiel 8, lassen das Training mit neuem Log-Ordner laufen und aktualisieren das Dashboard. Das geht durch einen Klick auf das "Neu laden" Symbol oben rechts. Wie du siehst, sind die entsprechenden Werte jetzt in der Grafik mit eingetragen worden.

Einblendung Tensorboard

Wir sehen gleich, dass der Fehler sowohl bei den Trainings- als auch bei den Validierungsdaten sehr hoch ist. Es ist also zu einer Unteranpassung gekommen. Auch die Dauer des Trainings kann Unter- und Überanpassung beeinflussen. Wird zu kurz trainiert, kann das Modell nicht alle relevanten Zusammenhänge lernen, wird zu lange trainiert, lernt das Modell das sogenannte Rauschen der Trainingsdaten mit, also kleine irrelevante Abweichungen. Um zu prüfen, ob die Dauer des Trainings etwas mit der festgestellten Unteranpassung zu tun hat, lassen wir unser kleines Modell viel länger laufen, z.B. mit 50

Epochen. Allerdings lässt sich trotzdem keine relevante Verbesserung feststellen. Das liegt daran, dass nicht die Dauer des Trainings das Problem ist, sondern die fehlende Komplexität unseres neuronalen Netzes.

Als nächstes gucken wir uns an, ob wir eine Überanpassung erzeugen können. Zur besseren Übersichtlichkeit lösche ich dafür die Logs der Unteranpassung. Angenommen, wir lassen das Standard-Modell mit 128 Knoten in der versteckten Schicht auch länger laufen, z.B. 50 Epochen. Der Fehler bei den Trainingsdaten ist nach der letzten Epoche winzig, nämlich bei 0.003, steigt aber bei den Validierungsdaten an.

Einblendung Tensorboard

Hier ist es also tatsächlich zur Überanpassung gekommen. Es wäre also besser gewesen, wir hätten einen Callback verwendet, der das Training vorzeitig beendet hätte, bevor der Fehler anstieg. Dieser Callback ist ebenfalls bereits im Modul Keras enthalten, heißt Early Stopping und wird so mit übergeben. Der Parameter monitor gibt an, auf was der Callback achten soll, in diesem Fall also den Fehler bei den Validierungsdaten. Der Parameter patience gibt an, wie viele Epochen sich der Fehler bei den Validierungsdaten nicht verbessern darf, bevor abgebrochen wird. Da eine Verbesserung manchmal einige Epochen auf sich warten lässt, kann es sinnvoll sein, diesen Parameter höher zu setzen. In unserem Fall haben wir ja schon gesehen, dass schon wenige Epochen zu viel zur Überanpassung führen, daher habe ich den Wert hier klein gesetzt. Wie wir hier sehen, wird das Training nach acht Epochen vorzeitig beendet. Der genaue Wert kann variieren, da der Optimierungsalgorithmus Adam mit Zufall arbeitet.

```
early_stopping_callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience = 2)
model.fit(x=x_train,
        y=y_train,
        epochs=8,
        validation_data=(x_val, y_val),
        callbacks=[tensorboard_callback, early_stopping_callback])
```

Ändern der Schrittweite

Jetzt zeige ich dir noch ein Beispiel, bei dem wir an den Einstellungen des Optimierungsalgorithmus herumspielen. Wir trainieren für zehn Epochen, einmal mit der Standardeinstellung, also einer Schrittweite von 0.001, einmal mit einem Fünftel davon, also 0.0002, und einmal mit dem Fünffachen der Standardeinstellung, also 0.005. Wenn wir den Fehler bei der Validierung betrachten, sehen wir gleich das unterschiedliche Verhalten der verschiedenen Schrittweiten. Die kleinere Schrittweite kommt langsamer zum Ziel als die Standardeinstellung, während die größere Schrittweite fasst sofort zu einer Steigerung des Fehlers führt. Das heißt aber nicht, dass eine höhere Schrittweite immer besser oder immer schlechter ist. Genau das macht das Optimieren ja zu einer so schweren Aufgabe.

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate = 0.001),
```

```
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate = 0.0002),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate = 0.005),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
```

Einblendung Tensorboard

Das Überprüfen der Metrik bei den Validierungsdaten hilft uns schon, abschätzen zu können, ob unser Training gut oder schlecht funktioniert. Wichtig ist aber der Erfolg bei den Testdaten, da das die Daten sind, die das Modell beim Training noch nicht gesehen hat. Dafür müssen wir unser Modell evaluieren. Auch hier kann wieder ein Tensorboard-Callback übergeben werden, ich verzichte aber dieses Mal darauf. Wir sehen, dass die Standardeinstellung die beste war, gefolgt von der kleineren Schrittweite. Da das Training bei kleinerer Schrittweite langsamer verläuft, kann es aber sein, dass sich das Ergebnis bei mehr als zehn Epochen noch deutlich verbessert.

```
model.evaluate(x_test, y_test)
```

Abschluss

In diesem Video hast du die Anwendung Tensorboard kennen gelernt und dir das Training deines Modells visualisieren lassen. Du kannst ein neuronales Netz trainieren und evaluieren und dabei Log-Dateien erstellen und anzeigen lassen. Außerdem hast du gelernt, was Callbacks sind und wie du sie einsetzen kannst.

Quellen

Quelle [1] Josef Steppan, CC BY-SA 4.0, via Wikimedia Commons
https://commons.wikimedia.org/wiki/File:MnistExamples.png

Weiterführendes Material

https://www.tensorflow.org/tensorboard/get_started

https://neptune.ai/blog/tensorboard-tutorial

Disclaimer