© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

KAPITEL 2

Bildklassifizierung mit PyTorch

Nachdem Sie PyTorch eingerichtet haben, werden Sie in den meisten Deep-Learning-Lehrbüchern mit jeder Menge Fachjargon überhäuft, bevor Sie etwas Interessantes tun. Ich versuche, das auf ein Minimum zu beschränken, und erarbeite mit Ihnen ein Beispiel, das leicht erweitert werden kann, wenn Sie sich mit der Arbeit in PyTorch besser zurechtfinden. Wir verwenden dieses Beispiel im gesamten Buch, um zu erfahren, wie man ein Modell debuggt (Kapitel 7) oder es in den Produktiveinsatz bringt (Kapitel 8).

Was wir von jetzt an bis zum Ende von Kapitel 4 entwickeln werden, ist ein *Bildklassifikator*. Neuronale Netzwerke werden üblicherweise als Bildklassifikatoren verwendet; das Netzwerk erhält ein Bild und wird etwas gefragt, das für uns eine einfache Frage darstellt: »Was ist das?«

Beginnen wir mit der Entwicklung unserer PyTorch-Anwendung.

Unsere Klassifizierungsaufgabe

Hier erstellen wir einen einfachen Klassifikator, der den Unterschied zwischen Fischen und Katzen erkennen soll. Wir werden das Design und die Art und Weise, wie wir unser Modell aufbauen, schrittweise verändern, um es immer akkurater zu gestalten.

Die Abbildungen 2-1 und 2-2 zeigen einen Fisch und eine Katze jeweils in ihrer ganzen Pracht. Ich bin mir nicht sicher, ob der Fisch einen Namen hat, aber die Katze heißt Helvetica.

Beginnen wir mit einer Erläuterung der traditionellen Herausforderungen, die mit der Klassifizierung verbunden sind.

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Abbildung 2-1: Ein Fisch!

Abbildung 2-2: Helvetica im Karton

Traditionelle Herausforderungen

Wie würden Sie ein Programm schreiben, das einen Fisch von einer Katze unterscheiden können soll? Vielleicht würden Sie eine Reihe von Regeln verfassen, die beschreiben, dass eine Katze einen Schwanz oder dass ein Fisch Schuppen hat, und diese Regeln auf ein Bild anwenden, um zu bestimmen, was Sie sehen. Aber das würde Zeit, Mühe und Geschicklichkeit erfordern. Und was passiert, wenn Sie auf etwas wie eine Manx-Katze stoßen? Obwohl es eindeutig eine Katze ist, besitzt sie keinen Schwanz.

Wie Sie sich vorstellen können, werden diese Regeln immer komplizierter, um sämtliche möglichen Szenarien zu beschreiben. Außerdem muss ich gestehen, dass

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

ich absolut schlecht in der Grafikprogrammierung bin, sodass mich allein der Gedanke, all diese Regeln manuell programmieren zu müssen, mit Furcht erfüllt.

Was wir suchen, ist eine Funktion, die bei der Eingabe eines Bilds entweder *Katze* oder *Fisch* zurückgibt. Diese Funktion ist für uns kaum zu programmieren, wenn wir alle Kriterien abschließend auflisten möchten. Durch Deep Learning wird jedoch der Rechner im Wesentlichen dazu gebracht, die harte Arbeit der Entwicklung all dieser Regeln zu leisten, von denen wir gerade gesprochen haben. Voraussetzung ist, dass wir eine Struktur schaffen, dem Netzwerk viele Daten geben sowie die Möglichkeit, herauszufinden, ob es die richtige Antwort liefert. Und genau das werden wir jetzt tun. Dabei werden Sie einige Schlüsselkonzepte für die Verwendung von PyTorch kennenlernen.

Zunächst erst mal Daten

Zunächst benötigen wir Daten. Wie viele Daten? Nun, das kommt darauf an. Die Vorstellung, dass man für jede Deep-Learning-Methode riesige Datenmengen benötigt, um das neuronale Netz zu trainieren, ist nicht unbedingt richtig, wie Sie noch in Kapitel 4 sehen werden. Für den Moment möchten wir jedoch ein Netzwerk von Grund auf trainieren, was meist den Zugriff auf eine große Datenmenge erfordert. Wir benötigen also eine Menge Bilder von Fischen und Katzen.

Wir könnten nun einige Zeit darauf verwenden, viele Bilder beispielsweise mithilfe der Google-Bildsuche herunterzuladen. In unserem Fall haben wir jedoch eine andere Lösung: eine Sammlung von Bildern namens *ImageNet*, die standardmäßig zum Trainieren neuronaler Netze verwendet wird. Sie enthält mehr als 14 Millionen Bilder und 20.000 Bildkategorien und ist der Standard zur Beurteilung der Leistungsfähigkeit aller Bildklassifikatoren. Ich nehme die Bilder von dort, aber Sie können auch gern selbst andere Bilder herunterladen, wenn Sie das bevorzugen.

Zusätzlich zu den Daten benötigt PyTorch eine Möglichkeit, mit der es bestimmen kann, was eine Katze und was ein Fisch ist. Das mag für uns leicht sein, aber für den Computer ist es nicht so einfach (deshalb bauen wir das Programm überhaupt erst!). Wir verwenden zur Kennzeichnung *Labels*, die den Daten beigefügt sind. Dieser Ansatz wird als *überwachtes Lernen* (engl. Supervised Learning) bezeichnet. (Wenn Sie keine Labels vorliegen haben, müssen Sie zum Trainieren wohl wenig überraschend auf *unüberwachtes Lernen* (engl. Unsupervised Learning) zurückgreifen.)

Wenn wir nun die ImageNet-Daten verwenden, werden die vorhandenen Labels nicht so nützlich sein, weil sie *zu viele* Informationen enthalten. Ein Label für *gestromte Katze* oder *Forelle* ist für den Computer getrennt zu betrachten vom Label *Katze* oder *Fisch*. Wir müssen die Bilder also neu labeln. Da ImageNet eine so umfangreiche Sammlung von Bildern ist, habe ich bereits eine Liste mit Bild-URLs und Labels (*https://oreil.ly/NbtEU*) für Fische und Katzen zusammengestellt.

Sie können das Skript *download.py* innerhalb Ihres aktuellen Arbeitsverzeichnisses ausführen. Es lädt die Bilder von den URLs herunter und speichert sie für das Trai-

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

ning in den entsprechenden Ordnern. Die *Kennzeichnung* mit neuen Labels gestaltet sich einfach; das Skript speichert Katzenbilder im Verzeichnis *train/cat* und Fischbilder in *train/fish*. Wenn Sie es vorziehen, das Skript nicht zum Herunterladen zu verwenden, erstellen Sie einfach die Verzeichnisse und speichern die entsprechenden Bilder an den richtigen Stellen. Jetzt haben wir unsere Daten zusammen. Allerdings müssen wir sie noch in ein Format bringen, das PyTorch verstehen kann.

Daten mit PyTorch einspielen

Das Laden und Konvertieren von Daten in trainingskonforme Formate gehört in der Datenwissenschaft oft zu den Bereichen, die viel zu viel unserer Zeit in Anspruch nehmen. PyTorch bietet Ihnen Standardkonventionen für die Interaktion mit Daten, die Ihnen, egal ob Sie mit Bild-, Text- oder Tondaten arbeiten, eine ziemlich konsistente Handhabung ermöglichen.

Die beiden wichtigsten Konventionen für den Umgang mit Daten sind die `Dataset`- und `DataLoader`-Klassen. `Dataset` ist eine Python-Klasse, die es uns erlaubt, an die Daten zu gelangen, die wir dem neuronalen Netz zur Verfügung stellen. Die Klasse `DataLoader` speist die Daten aus dem Datensatz in das Netzwerk ein. (Dies kann die Ausgabe von Informationen wie »Wie viele Arbeitsprozesse speisen Daten in das Netzwerk ein?« oder »Wie viele Bilder geben wir auf einmal ein?« umfassen.)

Sehen wir uns zunächst die `Dataset`-Klasse an. Jeder Datensatz, egal ob er Bilder, Audio, Text, dreidimensionale Landschaften, Börseninformationen oder was auch immer enthält, kann mit PyTorch interagieren, wenn er dieser abstrakten Python-Klasse genügt:

```
class Dataset(object):
    def __getitem__(self, index):
        raise NotImplementedError

    def __len__(self):
        raise NotImplementedError
```

Das ist ziemlich einfach: Wir müssen eine Methode implementieren, die die Größe unseres Datensatzes (`len`) zurückgibt, und eine Methode, mit der wir ein Element aus unserem Datensatz in Form eines (*`Label`*, *`Tensor`*)-Paars abrufen können. Dieses wird vom `DataLoader`-Objekt aufgerufen, um die Daten zum Training in das neuronale Netz zu schicken. Wir müssen also einen Funktionskörper für `getitem` schreiben, damit die Methode ein Bild aufnehmen, in einen Tensor transformieren und diesen sowie das Label zurückgeben kann, damit PyTorch darauf operieren kann. So weit, so gut, aber Sie können sich vorstellen, dass dieses Szenario durchaus häufig auftaucht. Vielleicht kann PyTorch die Dinge für uns ja einfacher gestalten?

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Einen Trainingsdatensatz erstellen

Das `torchvision`-Paket enthält eine Klasse namens `ImageFolder`, die so ziemlich alles für uns tut, vorausgesetzt, unsere Bilder befinden sich in einer Struktur, in der jedes Verzeichnis alle Bilder eines Labels enthält (z.B. alle Katzen befinden sich in einem Verzeichnis namens *cat*). Für unser Beispiel mit den Katzen und Fischen genügen uns etwa die Befehle in nachfolgendem Code. Um zu gewährleisten, dass unsere `DataLoader`-Objekte nur valide Bilder erhalten, integrieren wir ein weiteres Argument, `is_valid_file`, das mittels der Hilfsfunktion `check_image()` zunächst prüft, ob die Bilder problemlos verarbeitet werden können:[1]

```
import torchvision
from PIL import Image, ImageFile
from torchvision import transforms
from torchvision.datasets import ImageFolder

ImageFile.LOAD_TRUNCATED_IMAGES = True

def check_image(path):
    try:
        im = Image.open(path)
        return True
    except:
        return False

train_data_path = "./train/"

img_transforms = transforms.Compose([
                        transforms.Resize((64,64)),
                        transforms.ToTensor(),
                        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                             std=[0.229, 0.224, 0.225])
])

train_data = ImageFolder(root=train_data_path,
                         transform=img_transforms,
                         is_valid_file=check_image)
```

Darüber hinaus können Sie mit `torchvision` auch eine Reihe von Transformationen festlegen, die auf die Bilder angewendet werden, bevor sie in das neuronale Netz eingespeist werden. Die Standardtransformation besteht darin, die Bilddaten in einen Tensor umzuwandeln (die `transforms.ToTensor()`-Methode aus dem vorhergehenden Codeblock). Zusätzlich führen wir auch einige weitere Arbeitsschritte durch, die vielleicht nicht ganz offensichtlich sind.

GPUs sind so konzipiert, dass sie schnell Berechnungen durchführen können, die auf eine Standardgröße abzielen. Aber wahrscheinlich haben wir eine Auswahl an Bildern in vielen verschiedenen Auflösungen. Um unsere Rechenleistung zu erhö-

1 Sollte `torchvision` oder `PIL` noch nicht installiert sein, geben Sie einfach `conda install -c pytorch torchvision` bzw. `pip install torchvision` und für den Fork der PIL-Bibliothek, Pillow, `pip install Pillow` in Ihre Kommmandozeile ein.

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

hen, skalieren wir jedes eingehende Bild auf die gleiche Auflösung von 64 × 64 mittels der `Resize((64,64))`-Transformation. Anschließend konvertieren wir die Bilder in einen Tensor und normalisieren ihn um einen bestimmten Satz von Mittelwerten und Standardabweichungen.

Die Normalisierung ist wichtig, da beim Durchlaufen der Eingabe durch die Schichten des neuronalen Netzes viele Multiplikationen stattfinden; die Beschränkung der eingehenden Werte auf einen Bereich von 0 bis 1 verhindert, dass die Werte während des Trainings zu groß werden (bekannt als das Problem des *explodierenden Gradienten*). Dazu benötigt diese magische Umwandlung nur den Mittelwert und die Standardabweichung des ImageNet-Datensatzes als Ganzes. Man könnte sie speziell für unsere Fisch- und Katzen-Untergruppen berechnen, aber unsere Werte sind akzeptabel. (Würden Sie an einem völlig anderen Datensatz arbeiten, müssten Sie den Mittelwert und die Standardabweichung berechnen, wobei auch viele einfach auf diese ImageNet-Konstanten zurückgreifen und über akzeptable Ergebnisse berichten).

Die zusammenfügbaren Transformationen können wir ebenfalls problemlos für Dinge wie Bilddrehung und -krümmung zu Zwecken der Datenaugmentation nutzen, auf die wir in Kapitel 4 zurückkommen werden.

In diesem Beispiel ändern wir die Auflösung der Bilder auf 64 × 64. Ich habe diese willkürliche Wahl getroffen, um in unserem kommenden ersten Netzwerk eine schnellere Berechnung zu ermöglichen. Die meisten Architekturen, die Sie in Kapitel 3 sehen werden, verwenden eine Auflösung von 224 × 224 oder 299 × 299 für ihre Bildeingaben. Im Allgemeinen gilt: Je größer der Eingabewert, desto mehr Informationen kann das Netzwerk daraus lernen. Die Kehrseite der Medaille ist, dass man oft nur eine kleinere Menge von Bildern in dem Speicher der GPU vorhalten kann.

Wir sind noch nicht ganz fertig mit der Datenvorbereitung. Wir benötigen neben dem Trainingsdatensatz noch weitere Datensätze. Aber warum?

Erstellen eines Validierungs- und eines Testdatensatzes

Unsere Trainingsdaten sind fertig vorbereitet – wir müssen jetzt noch die gleichen Schritte für unseren *Validierungsdatensatz* wiederholen. Worin liegt der Unterschied? Eine Gefahr beim Deep Learning (und eigentlich bei allen Formen des maschinellen Lernens) ist das Phänomen der *Überanpassung* (engl. Overfitting): Ihr Modell wird wirklich gut darin, das korrekt zu erkennen, worauf es trainiert wurde, kann aber nicht auf Beispiele verallgemeinern, die es nicht gesehen hat. Es sieht also das Bild einer Katze, und wenn nicht alle anderen Bilder von Katzen diesem Bild sehr ähnlich sind, glaubt das Modell nicht, dass es sich um eine Katze handelt, obwohl es offensichtlich eine Katze ist. Um unser Netzwerk daran zu hindern, laden wir einen *Validierungsdatensatz* mithilfe der Datei *download.py* herunter, der eine Reihe von Katzen- und Fischbildern enthält, die nicht im Trainingsdatensatz enthalten sind.

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Am Ende eines jeden Trainingszyklus (auch als *Epoche* bekannt) vergleichen wir die Leistung mit diesem Datensatz, um sicherzustellen, dass unser Netzwerk keine Fehler macht. Aber keine Sorge – der Code dafür ist unglaublich einfach; es ist lediglich der vorherige Code mit ein paar geänderten Variablennamen:

```
val_data_path = "./val/"
val_data = ImageFolder(root=val_data_path,
                       transform=img_transforms,
                       is_valid_file=check_image)
```

Wir haben die vorherige Definition der verketteten `img_transforms` einfach erneut verwendet, anstatt sie noch einmal zu definieren.

Zusätzlich zum Validierungsdatensatz sollten wir auch einen *Testdatensatz* erstellen. Dieser dient der abschließenden Modellbewertung nach Beendigung des gesamten Trainings:

```
test_data_path = "./test/"
test_data = ImageFolder(root=test_data_path,
                        transform=img_transforms,
                        is_valid_file=check_image)
```

Die Unterscheidung der verschiedenen Datensätze kann etwas verwirrend sein, deshalb habe ich eine Tabelle zusammengestellt, die Ihnen Auskunft darüber gibt, welcher Datensatz in welchem Schritt in der Modellentwicklung verwendet wird (siehe Tabelle 2-1).

Tabelle 2-1: Arten von Datensätzen

Trainingsdatensatz	Wird während des Trainingsvorgangs zur Aktualisierung des Modells verwendet.
Validierungsdatensatz	Wird verwendet, um zu bewerten, wie gut das Modell auf die jeweilige Aufgabendomäne verallgemeinert, statt ausschließlich die Trainingsdaten abzubilden; wird nicht zur direkten Aktualisierung des Modells verwendet.
Testdatensatz	Ein finaler Datensatz, der eine abschließende Bewertung der Leistung des Modells nach Beendigung des Trainings liefert.

Mit ein paar weiteren Zeilen Python-Code können wir nun unsere `DataLoader`-Objekte erstellen:

```
from torch.utils.data import DataLoader

# Kann bei wenig Speicher auch kleiner gewählt werden, z.B. 32:
batch_size = 64

train_data_loader = DataLoader(train_data, batch_size=batch_size)
val_data_loader = DataLoader(val_data, batch_size=batch_size)
test_data_loader = DataLoader(test_data, batch_size=batch_size)
```

Das neue Element in diesem Code ist die Angabe der `batch_size` (Batchgröße). Diese sagt uns, wie viele Bilder durch das Netzwerk geschickt werden sollen, ehe wir es trainieren und aktualisieren. Theoretisch könnten wir die `batch_size` auf die Anzahl der Bilder in den Test- und Trainingsdatensätzen einstellen, sodass das Netzwerk jedes

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Bild sieht, bevor es aktualisiert wird. In der Praxis tendieren wir dazu, das nicht zu tun, weil kleinere Batches (in der Literatur eher als *Minibatches* bekannt) weniger Speicherplatz benötigen, als wenn wir *alle* Informationen über *jedes* Bild im Datensatz speichern müssten. Die kleinere Batchgröße führt letztendlich dazu, dass das Training rascher vonstattengeht, da wir unser Netzwerk viel schneller aktualisieren.

Standardmäßig sind die `DataLoader` von PyTorch auf eine `batch_size` von 1 eingestellt. Sie werden das mit ziemlicher Sicherheit ändern wollen. Obwohl ich hier 64 ausgewählt habe, sollten Sie dennoch damit experimentieren, um zu sehen, wie groß ein Minibatch sein kann, ohne den Speicher Ihrer GPU zu überlasten. Vielleicht möchten Sie auch mit einigen der zusätzlichen Parameter experimentieren: Sie können angeben, wie die Datensätze gesampelt werden, ob der komplette Datensatz bei jedem Durchlauf neu gemischt wird und wie viele Arbeitsprozesse genutzt werden, um Daten aus dem Datensatz zu entnehmen. All dies kann in der PyTorch-Dokumentation (*https://oreil.ly/XORs1*) nachgelesen werden.

So viel zum Einspeisen von Daten in PyTorch. Lassen Sie uns jetzt ein einfaches neuronales Netz entwickeln, um endlich mit der Klassifizierung unserer Bilder zu beginnen.

Endlich, ein neuronales Netzwerk!

Wir beginnen mit dem einfachsten Deep-Learning-Netzwerk: einer Eingabeschicht, die mit den Eingabetensoren (unseren Bildern) arbeitet, einer Ausgabeschicht, die so groß ist wie die Anzahl unserer Ausgabekategorien (2), und einer verborgene Schicht zwischen ihnen. In unserem ersten Beispiel verwenden wir vollständig verbundene Schichten. Abbildung 2-3 illustriert, wie das mit einer Eingabeschicht mit drei Neuronen, einer verborgenen Schicht mit drei Neuronen und unserer zwei Neuronen umfassenden Ausgabe aussieht.

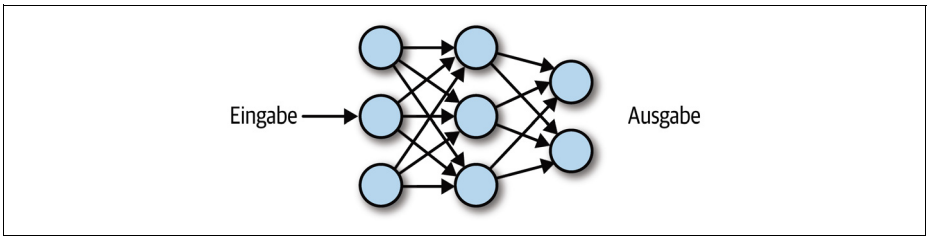

Abbildung 2-3: Ein einfaches neuronales Netzwerk

Wie Sie in diesem Beispiel mit vollständig verbundenen Neuronen sehen können, wirkt jedes Neuron in einer Schicht auf jedes Neuron der nächsten Schicht. Jede Verbindung besitzt ein *Gewicht*, das die Stärke des Signals von diesem Neuron bestimmt, das in die nächste Schicht geht (diese Gewichte werden im Rahmen des Trainings des Netzwerks üblicherweise infolge einer zufälligen Initialisierung aktualisiert). Wenn eine Eingabe das Netzwerk durchläuft, können wir (oder besser, PyTorch) einfach eine Matrixmultiplikation der Gewichte und des Bias-Werts die-

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

ser Schicht mit der Eingabe durchführen. Bevor es in die nächste Komponente geleitet wird, geht dieses Ergebnis in eine *Aktivierungsfunktion* ein. Auf diese Weise lässt sich relativ einfach Nichtlinearität in unser System integrieren.

Aktivierungsfunktionen

Aktivierungsfunktionen klingen zunächst kompliziert. Die heutzutage in der Literatur am häufigsten anzutreffende Aktivierungsfunktion wird als `ReLU`, genauer *Rectified Linear Unit*, bezeichnet. Was wiederum kompliziert klingt! Sie ist jedoch nicht mehr als eine Funktion, die *max(0,x)* implementiert, also als Ergebnis eine 0 liefert, wenn die Eingabe negativ ist, oder einfach die Eingabe (x), wenn x positiv ist. Wirklich simpel!

Eine weitere Aktivierungsfunktion, der Sie wahrscheinlich begegnen werden, heißt *Softmax*, die mathematisch etwas komplizierter ist. Im Grunde erzeugt sie eine Menge von Werten zwischen 0 und 1, die sich gemeinsam zu Wert 1 summieren (Wahrscheinlichkeiten!), und gewichtet die Werte so, dass sie Unterschiede überbewertet – das heißt, sie liefert einen Vektor, in dem ein Resultat höher als alle anderen ist. Sie werden oft sehen, dass er am Ende eines Netzwerks zu Klassifizierungszwecken verwendet wird, um sicherzustellen, dass das Netzwerk eine eindeutige Vorhersage darüber macht, zu welcher Kategorie es die Eingabe zugehörig sieht.

Nachdem wir alle diese Bausteine eingeführt haben, können wir mit dem Aufbau unseres ersten neuronalen Netzes beginnen.

Ein Netzwerk erstellen

Die Erstellung eines Netzwerks in PyTorch gestaltet sich sehr pythonisch. Unsere Klasse erbt von einer Klasse namens `torch.nn.Module`, und wir füllen die Methoden `__init__` und `forward` unseren Vorstellungen entsprechend aus:

```
import torch.nn as nn
import torch.nn.functional as F

class SimpleNet(nn.Module):

    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(12288, 84)
        self.fc2 = nn.Linear(84, 50)
        self.fc3 = nn.Linear(50,2)

    def forward(self, x):
        x = x.view(-1, 12288)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x))
        return x

simplenet = SimpleNet()
```

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Auch das ist nicht allzu kompliziert. Wir geben die erforderliche Struktur in `init()` vor. In diesem Fall initiieren wir unseren Superklassenkonstruktor und die drei vollständig verbundenen Schichten (in PyTorch `Linear` genannt, im Gegensatz zu `Dense` bei Keras). Die `forward()`-Methode beschreibt, wie die Daten sowohl im Rahmen des Trainings- als auch während des Vorhersagevorgangs (Inferenz) durch das Netzwerk strömen. Erinnern Sie sich? Zuerst müssen wir den dreidimensionalen Tensor (*x* und *y* plus drei Farbkanäle: Rot, Grün, Blau) in ein Bild umwandeln – in einen eindimensionalen Tensor –, sodass er in die erste `Linear`-Schicht eingespeist werden kann. Dies erreichen wir mit der `view()`-Funktion. Von da an können Sie sehen, dass wir die Schichten und die Aktivierungsfunktionen der Reihe nach einsetzen und schließlich eine `softmax`-Ausgabe erhalten, die uns unsere Vorhersage für ein Bild liefert.

Die Anzahl der Neuronen in den verborgenen Schichten ist etwas willkürlich gewählt. Eine Ausnahme davon stellt die letzte Schicht dar, bei der die zwei Ausgaben unseren beiden Kategorien, Katze oder Fisch, entsprechen. Im Allgemeinen möchten Sie, dass die Daten in Ihren Schichten *komprimiert* werden, wenn sie durch das Netzwerk hindurchströmen. Wenn bei einer Schicht z. B. 50 Eingänge zu 100 Ausgängen führen, könnte das Netzwerk *lernen*, einfach die 50 Verbindungen zu 50 von den insgesamt 100 Ausgängen weiterzuleiten und seine Arbeit als erledigt zu betrachten. Indem wir die Größe der Ausgabe in Bezug auf die Eingabe reduzieren, zwingen wir diesen Teil des Netzwerks dazu, eine Darstellung der ursprünglichen Eingabe mit weniger Ressourcen zu lernen, wodurch wir uns erhoffen, dass es einige Merkmale der Bilder erkennt, die für das zu lösende Problem wichtig sind, wie z. B. das Erkennen einer Flosse oder eines Schwanzes.

Wenn unser Netzwerk eine Vorhersage liefert, können wir diese mit dem tatsächlichen Label des Ausgangsbilds vergleichen, um zu sehen, ob die Vorhersage richtig war. Allerdings benötigen wir einen Weg, der es PyTorch erlaubt, nicht nur zu ermitteln, ob eine Vorhersage richtig oder falsch ist, sondern auch, wie falsch oder richtig sie ist. Hierfür bedarf es einer Verlustfunktion.

Verlustfunktionen

Verlustfunktionen gehören zu den wichtigsten Elementen einer erfolgreichen Deep-Learning-Lösung. PyTorch verwendet sie, um zu beurteilen, wie es das Netzwerk aktualisieren muss, um die gewünschten Ergebnisse zu erzielen.

Verlustfunktionen können so kompliziert oder einfach sein, wie Sie es wünschen. PyTorch wird standardmäßig mit einer umfassenden Sammlung von Funktionen ausgeliefert, die die meisten Anwendungen abdecken, auf die Sie wahrscheinlich stoßen werden. Dennoch können Sie natürlich Ihre eigenen Funktionen programmieren, falls Sie eine individuelle Lösung benötigen. Im vorliegenden Fall werden wir eine bereits integrierte Verlustfunktion namens `CrossEntropyLoss` (Kreuzentropieverlust) verwenden, die, wie bei uns vorliegend, für mehrkategoriale Klassifizierungsaufgaben empfohlen wird. Eine weitere Verlustfunktion, die Ihnen wahr-

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

scheinlich begegnen wird, nennt sich `MSELoss` (engl. *Mean Squared Error Loss*). Sie ist der gewöhnliche mittlere quadratische Verlust, den Sie für eine numerische Vorhersage heranziehen können.

Etwas, das man bei der `CrossEntropyLoss`-Verlustfunktion beachten muss, ist, dass sie auch eine `softmax()`-Funktion als Teil seiner Operation beinhaltet; deshalb ändern wir unsere `forward()`-Methode wie folgt:

```
def forward(self, x):
    # In einen eindimensionalen Vektor umwandeln.
    x = x.view(-1, 12288)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

Sehen wir uns nun an, wie die Schichten eines neuronalen Netzes während einer Trainingsschleife aktualisiert werden.

Optimierung

Beim Training eines Netzwerks werden die Daten durch das Netzwerk geleitet, wobei die Verlustfunktion dazu verwendet wird, die Differenz zwischen der Vorhersage und dem tatsächlichen Label zu quantifizieren. Diese Informationen werden anschließend rückwärts gerichtet durch das Netzwerk geleitet und zur Aktualisierung der Gewichte des Netzwerks verwendet, um zu versuchen, den Verlust so gering wie möglich zu halten. Zur Aktualisierung des neuronalen Netzes verwenden wir einen sogenannten *Optimierer*.

Für den Fall, dass wir nur ein Gewicht hätten, könnten wir einfach dem jeweiligen Verlustwert den Wert des Gewichts visuell nachvollziehbar gegenüberstellen. Der Zusammenhang könnte beispielsweise wie in Abbildung 2-4 aussehen.

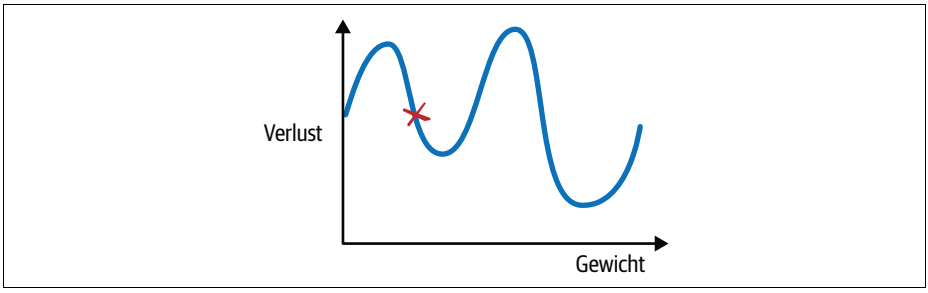

Abbildung 2-4: Zweidimensionale Darstellung einer Verlustfunktion

Nehmen wir als Ausgangspunkt eine zufällige Position in Abbildung 2-4, die durch das X markiert ist. Den entsprechenden Gewichtswert können wir auf der x-Achse und den Wert des Verlusts auf der y-Achse ablesen. Um unsere optimale Lösung zu erreichen, müssen wir zum niedrigsten Punkt auf der Kurve gelangen. Wir können uns entlang der Funktion bewegen, indem wir den Wert des Gewichts ändern und

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

so einen neuen Wert für die Verlustfunktion erhalten. Anhand der Steigung der Kurve können wir einschätzen, wie gut wir uns hinsichtlich der Erreichung des Optimums bewegen. Eine verbreitete Methode zur Visualisierung des Optimierers ist das Rollen einer Murmel, um den niedrigsten Punkt (bzw. das *Minimum*) in einer bestimmten Anzahl von Tälern zu finden. Dies wird vielleicht deutlicher, wenn wir unser Beispiel auf zwei Parameter erweitern und so ein dreidimensionales Diagramm wie das in Abbildung 2-5 gezeigte erhalten.

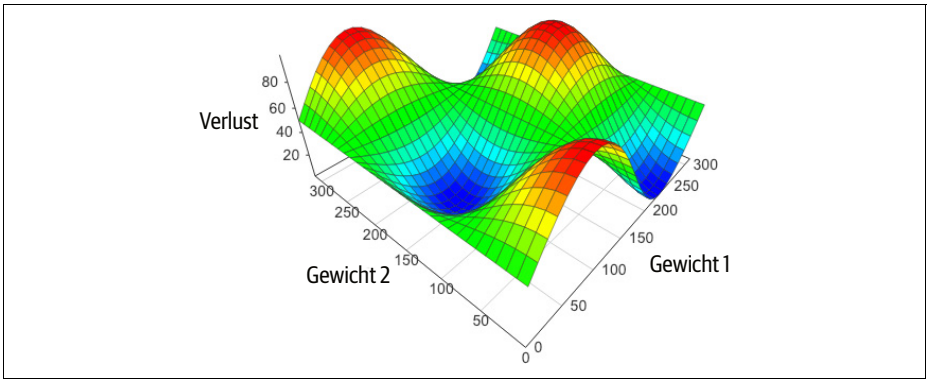

Abbildung 2-5: Dreidimensionale Darstellung einer Verlustfunktion

In diesem Fall können wir an jedem Punkt die Gradienten aller potenziellen Bewegungen ermitteln und denjenigen auswählen, der uns am deutlichsten den Berg hinunterbringt.

Allerdings sollten Sie sich über ein paar Dinge im Klaren sein. Zum einen besteht die Gefahr, sich in *lokalen Minima* zu verfangen, d. h. in Bereichen, die im Rahmen der Ermittlung unserer Gradienten so aussehen, als seien sie die tiefsten Teile der Verlustkurve, obwohl in Wirklichkeit jedoch tiefer liegende Bereiche existieren. Anhand unserer eindimensional verlaufenden Kurve in Abbildung 2-4 können wir sehen, dass wir, wenn wir durch kleine Sprünge nach unten in einem der Minima auf der linken Seite landen, nie einen Grund hätten, diesen Bereich wieder zu verlassen. Wenn wir größere Sprünge machen, könnten wir auf den Weg zum eigentlichen Minimum gelangen. Da wir allerdings immer wieder so große Sprünge machen, überspringen wir auch immer wieder den ganzen Bereich.

Die Weite unserer Sprünge wird als *Lernrate* bezeichnet und ist oft der *entscheidende* zu justierende Parameter, der dazu beiträgt, dass Ihr Netzwerk richtig und effizient lernen kann. In Kapitel 4 werden Sie eine weitere Methode kennenlernen, um eine geeignete Lernrate zu bestimmen. Vorerst experimentieren wir einfach mit unterschiedlichen Werten: Versuchen wir es zunächst einmal mit einem Wert wie 0,001. Wie eben erwähnt, führen große Lernraten dazu, dass Ihr Netzwerk während des Trainings über den ganzen Bereich springt, ohne dass es zu einer *Konvergenz* der Gewichte kommt.

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Um das Problem der lokalen Minima zu entschärfen, nehmen wir eine leichte Änderung bei der Ermittlung aller möglichen Gradienten vor und stellen zufällig gesampelte Gradienten während eines Batches bereit. Diese Methode wird als *stochastischer Gradientenabstieg* (engl. *Stochastic Gradient Descent* (SGD)) bezeichnet und stellt den traditionellen Ansatz zur Optimierung neuronaler Netze und anderer maschineller Lernverfahren dar. Es gibt noch weitere Optimierungsalgorithmen, die im Bereich des Deep Learning vorzuziehen sind. PyTorch wird standardmäßig mit SGD sowie anderen wie AdaGrad, RMSProp und Adam, dem Optimierer, den wir für den Großteil des Buchs verwenden werden, ausgeliefert.

Eine der wichtigsten Verbesserungen, die Adam (wie auch RMSProp und AdaGrad) vornimmt, besteht darin, dass er jeweils pro Parameter eine Lernrate verwendet und diese in Abhängigkeit von der Änderungsrate dieser Parameter anpasst. Dabei wird eine exponentiell abnehmende Liste von Gradienten und der Quadrate dieser Gradienten gespeichert und dazu verwendet, die globale Lernrate, mit der Adam arbeitet, zu skalieren. Es hat sich empirisch gezeigt, dass Adam die meisten anderen Optimierer in tiefen Netzwerken übertrifft. Sie können Adam aber auch gegen SGD, RMSProp oder einen weiteren Optimierer austauschen, um nachzuvollziehen, ob der Einsatz einer anderen Methode ein schnelleres und verbessertes Training für Ihre spezielle Anwendung ermöglicht.

Die Erstellung eines auf Adam basierenden Optimierers ist einfach. Wir rufen die Funktion `optim.Adam()` auf und übergeben die Gewichte des zu aktualisierenden Netzwerks (die wir mithilfe der Funktion `simplenet.parameters()` erhalten) sowie unsere exemplarisch gewählte Lernrate von 0,001:

```
import torch.optim as optim
optimizer = optim.Adam(simplenet.parameters(), lr=0.001)
```

Der Optimierer ist das letzte fehlende Teil des Puzzles – wir können also endlich anfangen, unser Netzwerk zu trainieren.

Training

Widmen wir uns nun unserer vollständigen Trainingsschleife, die alles, was Sie bisher kennengelernt haben, für das Training des Netzwerks kombiniert. Wir werden diese als Funktion formulieren, damit wir Komponenten wie die Verlustfunktion und den Optimierer als Parameter übergeben können. An diesem Punkt sieht es zunächst recht allgemein aus:

```
for epoch in range(epochs):
    model.train()
    for batch in train_loader:
        optimizer.zero_grad()
        inputs, targets = batch
        output = model(inputs)
        loss = loss_fn(output, targets)
        loss.backward()
        optimizer.step()
```

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Auch wenn der Code ziemlich unkompliziert ist, sollten Sie dennoch ein paar Dinge beachten. Wir nehmen bei jeder Iteration der Schleife ein Batch aus unserem Trainingsdatensatz, der von unserem `DataLoader`-Objekt verarbeitet wird. Dieses lassen wir durch unser Modell laufen und berechnen den Verlust auf Basis der ermittelten Ausgabe. Dann rufen wir die `backward()`-Methode auf, um die Gradienten auf Grundlage dieses Verlusts im Rahmen der sogenannten Backpropagation, auch Fehlerrückführung genannt, zu berechnen. Die Methode `optimizer.step()` verwendet diese Gradienten anschließend, um die im vorherigen Abschnitt erwähnte Anpassung der Gewichte durchzuführen.

Was aber bewirkt der Funktionsaufruf `zero_grad()`? Bei genauerem Hinsehen stellt sich heraus, dass die berechneten Gradienten standardmäßig kumuliert werden (engl. accumulated). Das bedeutet, dass die Gradienten am Ende der Iteration eines Batches nicht auf null gesetzt werden, sondern dass beim nächsten Batch die Gradienten dieses Batches gemeinsam mit den eigentlichen Gradienten verarbeitet werden müssten und beim darauffolgenden Batch die beiden vorhergehenden und so weiter. Dies erweist sich nicht als hilfreich, da wir für unsere Optimierung in jeder Iteration nur die Gradienten des aktuellen Batches berücksichtigen möchten. Deshalb rufen wir die Funktion `zero_grad()` auf, um sicherzustellen, dass die Gradienten auf null zurückgesetzt werden, nachdem wir mit unserer Schleife fertig sind.

Im obigen Code ist Ihnen wahrscheinlich auch der Funktionsaufruf `model.train()` aufgefallen. Sie erfahren gleich noch mehr darüber, wenn wir uns den Aufbau der Validierungsschleife ansehen.

So weit zur Kurzfassung der Trainingsschleife. Wir müssen jedoch noch ein paar Dinge genauer unter die Lupe nehmen, ehe wir unsere komplette Funktion formulieren können.

Validierung

Sobald wir die Parameter unseres Modells nach dem Training mit dem gesamten Datensatz für die aktuelle Epoche aktualisiert haben, sollten wir das Modell anhand des Validierungsdatensatzes, den wir zuvor erstellt haben, hinsichtlich seiner Leistung auf einem anderen Datensatz bewerten. Dadurch stellen wir sicher, dass sich das Modell nicht zu sehr an den Trainingsdatensatz anpasst. Unsere Validierungsschleife sollte Ihnen bereits recht vertraut erscheinen:

```
model.eval()
for batch in val_loader:
    inputs, targets = batch
    output = model(inputs)
    loss = loss_fn(output,targets)
```

Wir leiten den Datensatz erneut Batch für Batch durch das Modell, vergleichen die Vorhersagen mit den tatsächlichen Labels des Batches und berechnen darauf basie-

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

rend den Verlust. Da die Parameter des Modells im Rahmen der Validierung nicht aktualisiert werden, benötigen wir weder die `backward()`-Methode noch eine Optimierungsroutine.

Doch was bewirkt der Aufruf der Methode `model.eval()`? Ein Modell kann sich in einem von zwei Modi befinden: dem *Trainingsmodus*, der durch den Aufruf von `model.train()` gesetzt wird, oder dem Evaluierungs- bzw. *Inferenzmodus*, der durch `model.eval()` aufgerufen wird. Wie Sie noch im Verlauf des Buchs sehen werden, können sich die Schichten eines Modells je nach Modus unterschiedlich verhalten. Wenn Sie also in einem anderen Zusammenhang als dem Modelltraining Vorhersagen treffen möchten, sollten Sie vorher *immer* `model.eval()` aufrufen.

Ein Modell auf der GPU zum Laufen bringen

Wie Sie vielleicht bereits bemerkt haben, läuft der Code nicht besonders schnell. Wie sähe es mit der leistungsstarken GPU aus, die in unserer Instanz in der Cloud verfügbar ist (oder dem sehr teuren Desktoprechner, den Sie sich zusammengestellt haben)? PyTorch führt standardmäßig CPU-basierte Berechnungen durch. Um die Vorteile der GPU zu nutzen, müssen wir unsere Eingabetensoren und das Modell selbst auf die GPU verschieben, indem wir explizit die Methode `to()` verwenden. Hier ein Beispiel, bei dem das `SimpleNet` auf die GPU kopiert wird:

```
import torch

if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

simplenet.to(device)
# Allgemein: model.to(device)
```

Sofern PyTorch ausgibt, dass eine GPU verfügbar ist, wird das Modell auf diese kopiert. Ansonsten wird das Modell auf der CPU belassen. Durch diese Vorgehensweise können wir direkt am Anfang unseres Codes ermitteln, ob eine GPU verfügbar ist, und anschließend einfach `tensor|model.to(device)` für den Rest des Programms verwenden, da wir bereits sichergestellt haben, dass sich das Modell an der richtigen Stelle befindet.

In früheren Versionen von PyTorch würden Sie stattdessen die Methode `cuda()` verwenden, um die Daten auf die GPU zu kopieren. Wenn Sie in fremden Codes auf diese Methode stoßen, seien Sie sich einfach darüber bewusst, dass sie dasselbe tut wie die `to()`-Methode!

Damit sind nun alle für das Training erforderlichen Schritte abgeschlossen. Wir sind fast fertig!

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Alles in einem

Sie haben in diesem Kapitel bereits eine Menge verschiedener Codestücke gesehen. Machen wir uns ans Werk und konsolidieren wir sie. Wir fügen alles zusammen, um eine generische Trainingsmethode zu erstellen, die ein Modell sowie die Trainings- und Validierungsdaten zusammen mit den Angaben zur Lernrate und zur Batchgröße annimmt und das Training mit diesem Modell durchführt. Den folgenden Code verwenden wir im gesamten restlichen Buch:

```
def train(model, optimizer, loss_fn, train_loader, val_loader,
          epochs=20, device="cpu"):
    for epoch in range(epochs):
        training_loss = 0.0
        valid_loss = 0.0
        model.train()
        for batch in train_loader:
            optimizer.zero_grad()
            inputs, targets = batch
            inputs = inputs.to(device)
            targets = targets.to(device)
            output = model(inputs)
            loss = loss_fn(output, targets)
            loss.backward()
            optimizer.step()
            training_loss += loss.data.item() * inputs.size(0)
        training_loss /= len(train_loader.dataset)

        model.eval()
        num_correct = 0
        num_examples = 0
        for batch in val_loader:
            inputs, targets = batch
            inputs = inputs.to(device)
            output = model(inputs)
            targets = targets.to(device)
            loss = loss_fn(output,targets)
            valid_loss += loss.data.item() * inputs.size(0)
            correct = torch.eq(torch.max(F.softmax(output, dim=1), dim=1)[1],
                               targets)
            num_correct += torch.sum(correct).item()
            num_examples += correct.shape[0]
        valid_loss /= len(val_loader.dataset)

        print('Epoch: {}, Training Loss: {:.2f}, Validation Loss: {:.2f}, '
              'accuracy = {:.2f}'.format(epoch+1, training_loss,
                                         valid_loss, num_correct/num_examples))
```

Es gibt auch ein paar neue Dinge, die im Code zu finden sind. Wir möchten den Trainingsverlust, den Validierungsverlust und die Modellgenauigkeit (`num_correct/num_examples`) jeweils für eine komplette Epoche berechnen. Für die Berechnung der Verluste geben wir nach jeder Epoche einen durchschnittlichen Verlust für die jeweilige Epoche aus. Dazu summieren wir einfach alle Verluste, die jeweils für ein

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Batch berechnet wurden, und teilen diese Summe durch die Gesamtgröße des Trainings- bzw. Validierungsdatensatzes, sobald am Ende einer Epoche alle Batches das Modell durchlaufen haben.

Die Berechnung der Modellgenauigkeit (engl. Accuracy), auch Relevanz genannt, ist ein wenig komplizierter. Zur Ermittlung müssen wir feststellen, wie viele richtige Vorhersagen das Modell in Bezug auf den Validierungsdatensatz trifft, und diesen Wert durch die Gesamtanzahl der getätigten Vorhersagen teilen. Hierzu müssen wir einfach den Vorhersagetensor durchgehen und einen Wert nach dem anderen mit dem tatsächlichen Label vergleichen. Mit PyTorch lässt sich diese Überprüfung jedoch bequemerweise mit dem gesamten Batch auf einmal durchführen; die Formulierung erscheint nur etwas komplex:

```
correct = torch.eq(torch.max(F.softmax(output, dim=1), dim=1)[1],
                   targets)
```

Für eine einzelne umbrochene Zeile Code geht hier eine Menge vor sich! Gehen wir sie Schritt für Schritt durch. Wir haben zunächst unseren `output`-Tensor mit den Vorhersagen unseres Modells, den wir einer `softmax()`-Aktivierungsfunktion übergeben (denken Sie daran, dass wir die `softmax()`-Funktion zuvor aus unserer `forward()`-Methode wieder entfernt haben). Aber was bewirkt der erste Parameter `dim=1`? Erinnern Sie sich kurz zurück, wie die `softmax`-Aktivierungsfunktion arbeitet: Sie nimmt Eingaben entgegen und bildet diese Eingabewerte auf einen neuen Satz von Ausgabewerten ab, die sich zu 1 summieren. Zum `dim`-Parameter müssen wir uns folgende Frage stellen: »Hinsichtlich welcher Dimension sollen sich die Werte des Tensors zu 1 summieren?« Wir möchten gewährleisten, dass sich die Werte als Wahrscheinlichkeiten in Bezug auf unsere beiden Kategorien interpretieren lassen, die in der Summe 1 ergeben. Dementsprechend muss sich diese Dimension auf unsere vorhergesagten Kategorien beziehen, weshalb wir `dim=1` angeben. (Würden wir `dim=0` wählen, würden sich die Werte in Bezug auf die Dimension des Batches zu 1 aufsummieren, d. h. jeweils alle Werte für die Bilder eines Batches hinsichtlich einer Kategorie, was wir mit ziemlicher Sicherheit nicht wollen).

Dann verwenden wir die Funktion `torch.max()`, um jeweils die Maximalwerte des Tensors, den wir durch die Anwendung der `softmax`-Aktivierungsfunktion erhalten, zu ermitteln. Hierzu geben wir wiederum `dim=1` als zusätzlichen Parameter an, um das Maximum entlang der Dimension der vorhergesagten Kategorie zu erhalten. Die `torch.max()`-Funktion gibt ein Tupel zurück, wobei das erste Element ein Tensor mit den Maximalwerten und das zweite ein Tensor mit den Werten der Indizes ist, an denen jeweils der Maximalwert gefunden wurde. Wir benötigen das zweite Element, da der Index des Maximalwerts unserer vorhergesagten Kategorie entspricht. Folglich verwenden wir `[1]`, um den Tensor mit den Indizes zu erhalten. Anschließend vergleichen wir diese Vorhersagen mit den tatsächlichen Labels, indem wir die vorhergesagten Werte dem Tensor `targets`, der unsere tatsächlichen Labels enthält, mithilfe der Funktion `torch.eq()` gegenüberstellen und bei Übereinstimmung der Werte den Wert `True` (d. h., die Vorhersage ist korrekt) und ande-

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

renfalls den Wert `False` erhalten. Gegebenenfalls müssten Sie den Tensor an dieser Stelle zusätzlich noch unter Verwendung von `view(-1)` so umformen, dass Sie einen eindimensionalen Tensor erhalten. Mit der folgenden Codezeile berechnen wir die Quersumme dieses Tensors und bilden die Summe für alle Batches; der Wert `False` entspricht dem Wert 0 und `True` dem Wert 1, sodass wir dadurch die Anzahl der korrekten Vorhersagen erhalten:

```
num_correct += torch.sum(correct).item()
```

Nun haben wir unsere Trainingsfunktion definiert – wir können umgehend mit dem Training beginnen, indem wir sie mit den erforderlichen Parametern aufrufen:

```
train(simplenet, optimizer, nn.CrossEntropyLoss(),
      train_data_loader, val_data_loader,
      epochs=20, device=device)
```

Das Netzwerk wird für 20 Epochen trainiert. Dies können Sie verändern, indem Sie einen anderen Wert für `epochs` in der Funktion `train()` angeben (hier hätten wir ihn auch weglassen können), und Sie sollten am Ende einer jeden Epoche eine Ausgabe zur Genauigkeit des Modells auf dem Validierungsdatensatz erhalten.

Sie haben Ihr erstes neuronales Netzwerk trainiert – herzlichen Glückwunsch! Jetzt können Sie damit Vorhersagen erstellen. Werfen wir also einen Blick darauf, wie das funktioniert.

Vorhersagen treffen

Schon zu Beginn des Kapitels hatte ich angekündigt, dass wir ein neuronales Netz erstellen würden, das dazu in der Lage ist, zu unterscheiden, ob auf einem Bild eine Katze oder ein Fisch abgebildet ist. Genau zu diesem Zweck haben wir inzwischen unser Netzwerk trainiert. Aber wie nutzen wir es, um uns eine Vorhersage für ein einzelnes Bild ausgeben zu lassen? Hier ist ein kurzer Abschnitt Python-Code, mit dem wir ein Bild aus dem Dateisystem laden und uns ausgeben lassen, ob unser Netzwerk *Katze* oder *Fisch* vorhersagt:

```
labels = ['cat','fish']

img = Image.open(FILENAME)
img = img_transforms(img).to(device)
img = torch.unsqueeze(img, 0)

simplenet.eval()
prediction = F.softmax(simplenet(img), dim=1)
prediction = prediction.argmax()
print(labels[prediction])
```

Der Großteil dieses Codes ist einfach. Wir verwenden wieder die zuvor erstellte Transformationspipeline, um das Bild in die richtige Form für unser neuronales Netzwerk zu überführen. Da unser Netzwerk jedoch Batches verwendet, erwartet es eigentlich einen vierdimensionalen Tensor, wobei sich die erste Dimension auf

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

die verschiedenen Bilder innerhalb eines Batches bezieht. Ein einzelnes Bild stellt zwar kein Batch dar, aber wir können mit diesem einzelnen Bild ein Batch der Länge 1 erzeugen, indem wir die Funktion `unsqueeze(img, 0)` verwenden, die eine neue Dimension an der ersten Stelle unseres Tensors hinzufügt. Wenn Sie Vorhersagen für mehrere Bilder erhalten möchten, können Sie die Klasse `torchvision.datasets.ImageFolder` einsetzen, um einen weiteren Datensatz zu erzeugen und den DataLoader, der die Batches liefert, ohne die `unsqueeze()`-Funktion nutzen zu können.

Nachdem wir das Modell in den Inferenzmodus gesetzt haben, gestaltet sich die Ermittlung von Vorhersagen ähnlich einfach. Wir müssen schlicht die Kategorie mit der höheren Wahrscheinlichkeit ermitteln. Im vorliegenden Fall könnten wir den Tensor einfach in eine Python-Liste umwandeln und die beiden Elemente vergleichen. Meistens gibt es jedoch mehr als nur zwei zu vergleichende Elemente. Dann erweist sich die PyTorch-Funktion `argmax()` als hilfreich, die den Index des höchsten Werts des Tensors zurückgibt. Diesen Index verwenden wir dann, um uns aus unserer Liste mit den Labels die entsprechende Vorhersage ausgeben zu lassen. Als Übung können Sie auf Basis der vorhergehenden Codeabschnitte die Vorhersagen für den kompletten Testdatensatz, den wir zu Beginn des Kapitels erstellt haben, ermitteln. Orientieren Sie sich insbesondere an der Validierungsschleife, um Vorhersagen für den Testdatensatz und das zugehörige `test_data_loader`-Objekt zu ermitteln, die wir zu Beginn des Kapitels erstellt haben. Auch hier brauchen Sie nicht die `unsqueeze()`-Funktion zu verwenden, da Sie vom `test_data_loader` bereits Batches in der korrekten Tensorform erhalten. Vielleicht hilft es Ihnen in diesem Zusammenhang, die Dimensionen der Tensoren mithilfe der Funktionen `shape` bzw. `size()` nachzuvollziehen.

Für den Moment ist das so ziemlich alles, was Sie über die Ermittlung von Vorhersagen wissen müssen; in Kapitel 8 kommen wir noch einmal darauf zurück, um das Ganze für den Produktiveinsatz aufzubereiten.

Zusätzlich zur Ermittlung der Vorhersagen würden wir wahrscheinlich gern in der Lage sein, das Modell zu jedem Zeitpunkt in der Zukunft mit unseren trainierten Parametern neu zu laden. Sehen wir uns an, wie das mit PyTorch realisiert wird.

Speichern von Modellen

Wenn Sie mit der Performance eines Modells zufrieden sind oder aus irgendeinem Grund eine Pause einlegen müssen, können Sie den aktuellen Zustand eines Modells im Python-Format *pickle* speichern, indem Sie die Methode `torch.save()` verwenden. Umgekehrt können Sie eine zuvor gespeicherte Iteration eines Modells laden, indem Sie die `torch.load()`-Methode verwenden.

Um unsere aktuellen Parameter und die Modellstruktur zu speichern, könnten wir also wie folgt vorgehen:

```
torch.save(simplenet, "/tmp/simplenet")
```

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Und so können Sie sie erneut laden:

```
simplenet = torch.load("/tmp/simplenet")
```

Dabei werden sowohl die Parameter als auch die Struktur des Modells in einer Datei gespeichert. Wenn Sie die Struktur des Modells zu einem späteren Zeitpunkt ändern, kann das ein Problem darstellen. Aus diesem Grund ist es üblicher, stattdessen das `state_dict` eines Modells zu speichern. Dies ist ein gewöhnliches Python-Dictionary (`dict`), das die Parameter aller Schichten im Modell umfasst. Das Speichern des `state_dict` gestaltet sich folgendermaßen:

```
torch.save(model.state_dict(), PATH)
```

In unserem Fall:

```
torch.save(simplenet.state_dict(), "/tmp/simplenet.pth")
```

Um es wiederherzustellen, erstellen Sie zuerst eine Instanz des Modells und verwenden dann `load_state_dict`. Für `SimpleNet` sähe das wie folgt aus:

```
simplenet = SimpleNet()
simplenet_state_dict = torch.load("/tmp/simplenet.pth")
simplenet.load_state_dict(simplenet_state_dict)
```

Der Vorteil bei diesem Vorgehen liegt darin, dass Sie, wenn Sie das Modell in irgendeiner Weise erweitern, einen `strict=False`-Parameter an `load_state_dict` übergeben können, der den Schichten im Modell Parameter zuweist, die im `state_dict` vorhanden sind. Dabei tritt kein Fehler auf, wenn das geladene `state_dict` Schichten umfasst, die in der aktuellen Struktur des Modells fehlen oder hinzugefügt wurden. Da es sich um ein normales Python-Dictionary handelt, können Sie die Namen der Schlüssel so ändern, dass sie zu Ihrem Modell passen. Das kann vor allem dann nützlich sein, wenn Sie Parameter aus einem völlig anderen Modell einlesen möchten.

Modelle können während eines Trainingslaufs auf einer Festplatte gespeichert und an einer anderen Stelle wieder geladen werden, sodass das Training dort fortgesetzt werden kann, wo Sie aufgehört haben. Das ist besonders nützlich, wenn man etwas wie Google Colab verwendet, das einen kontinuierlichen Zugriff auf eine GPU für nur etwa zwölf Stunden ermöglicht. Wenn Sie die Zeit im Auge behalten, können Sie das Modell vor der Unterbrechung speichern und das Training in einer neuen Zwölf-Stunden-Sitzung fortsetzen.

Zusammenfassung

Sie haben im Schnelldurchlauf die Grundlagen neuronaler Netze kennengelernt und erfahren, wie Sie diese in PyTorch mit einem Datensatz trainieren, wie Sie Vorhersagen für andere Bilder vornehmen und wie Sie Modelle auf und von der Festplatte speichern bzw. wiederherstellen können.

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40

Bevor Sie zum nächsten Kapitel übergehen, experimentieren Sie ruhig noch ein wenig mit der `SimpleNet`-Architektur, die wir entwickelt haben. Verändern Sie die Anzahl der Parameter in den `Linear`-Schichten und fügen Sie vielleicht noch eine oder zwei weitere Schichten hinzu. Schauen Sie sich die verschiedenen in PyTorch verfügbaren Aktivierungsfunktionen an und tauschen Sie `ReLU` gegen eine andere aus. Achten Sie darauf, was mit dem Training passiert, wenn Sie die Lernrate anpassen oder den Optimierer von Adam auf einen anderen umstellen (versuchen Sie es gern mit dem gewöhnlichen SGD). Ändern Sie gegebenenfalls die Batchgröße und die anfängliche Größe des Bilds, wenn es zu Beginn des Durchlaufs in einen eindimensionalen Tensor umgewandelt wird. Vieles im Deep Learning befindet sich noch in einer handwerklichen Konstruktionsphase; die Lernraten werden so lange von Hand ausgetüftelt, bis ein Netzwerk hinreichend trainiert ist. Es ist also durchaus ratsam, ein Verständnis vom Zusammenspiel aller adjustierbaren Elemente zu bekommen.

Sie mögen von der Genauigkeit der `SimpleNet`-Architektur vielleicht ein wenig enttäuscht sein – aber keine Sorge! In Kapitel 3 werden sich einige nennenswerte Verbesserungen einstellen, da wir anstelle des sehr einfachen Netzwerks, das wir bisher verwendet haben, ein neuronales Konvolutionsnetz einführen.

Weiterführende Literatur

- PyTorch-Dokumentation (*https://oreil.ly/x6pO7*)
- »Adam: A Method for Stochastic Optimization« (*https://arxiv.org/abs/1412.6980*) von Diederik P. Kingma und Jimmy Ba (2014)
- »An Overview of Gradient Descent Optimization Algorithms« (*https://arxiv.org/abs/1609.04747*) von Sebastian Ruder (2016)

© Dies ist urheberrechtlich geschütztes Material. Bereitgestellt von: UB Stuttgart Mi, Mai 18th 2022, 15:40