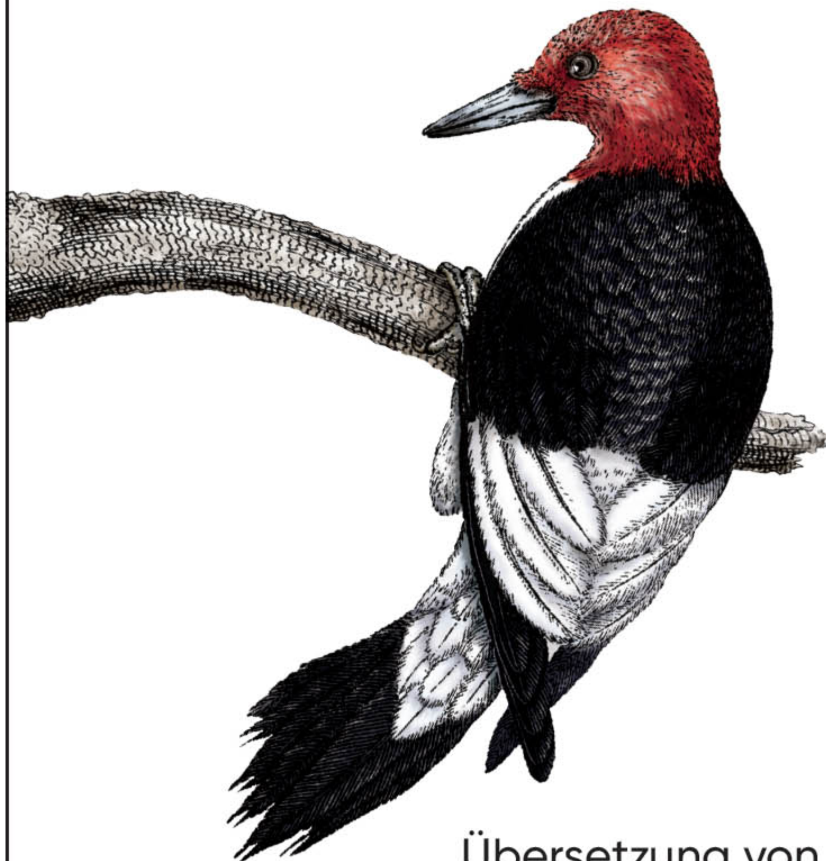
O'REILLY®

Deutsche Ausgabe

PyTorch für Deep Learning

Anwendungen für Bild-, Ton- und Textdaten entwickeln und deployen



Ian Pointer

Übersetzung von Marcus Fraaß

Inhalt

Cover

Titel

Impressum

Inhalt

Vorwort

1 Einstieg in PyTorch

- Zusammenbau eines maßgeschneiderten Deep-Learning-Rechners
  - Grafikprozessor (GPU)
  - Hauptprozessor (CPU) und Motherboard
  - Arbeitsspeicher (RAM)
  - Speicher
- Deep Learning in der Cloud
  - Google Colaboratory
  - Cloud-Anbieter
  - Welchen Cloud-Anbieter sollte ich wählen?
- Verwendung von Jupyter Notebook
- PyTorch selbst installieren
  - CUDA downloaden
  - Anaconda
  - Zu guter Letzt – PyTorch (und Jupyter Notebook)
- Tensoren
  - Tensoroperationen
  - Tensor-Broadcasting
- Zusammenfassung
- Weiterführende Literatur

2 Bildklassifizierung mit PyTorch

- Unsere Klassifizierungsaufgabe
- Traditionelle Herausforderungen
  - Zunächst erst mal Daten
  - Daten mit PyTorch einspielen
  - Einen Trainingsdatensatz erstellen
  - Erstellen eines Validierungs- und eines Testdatensatzes
- Endlich, ein neuronales Netzwerk!
  - Aktivierungsfunktionen
  - Ein Netzwerk erstellen
  - Verlustfunktionen
  - Optimierung
- Training
  - Validierung
  - Ein Modell auf der GPU zum Laufen bringen
- Alles in einem
  - Vorhersagen treffen
  - Speichern von Modellen
- Zusammenfassung
- Weiterführende Literatur

3 Neuronale Konvolutionsnetze (CNNs)

- Unser erstes Konvolutionsnetz
  - Konvolutionen
  - Pooling
  - Die Dropout-Schicht
- Die Geschichte der CNN-Architekturen
  - AlexNet
  - Inception/GoogLeNet

VGG

ResNet

Weitere Architekturen

Vortrainierte Modelle in PyTorch nutzen

Die Struktur eines Modells untersuchen

Die Batch-Normalisierungs-Schicht

Welches Modell sollten Sie verwenden?

One-Stop-Shopping für Modelle: PyTorch Hub

Zusammenfassung

Weiterführende Literatur

4 Transfer Learning und andere Kniffe

Transfer Learning mit ResNet

Die optimale Lernrate finden

Differenzielle Lernraten

Datenaugmentation

Transformationen in Torchvision

Farbräume und Lambda-Transformationen

Benutzerdefinierte Transformationsklassen

Klein anfangen und schrittweise vergrößern!

Ensemble-Modelle

Zusammenfassung

Weiterführende Literatur

5 Textklassifizierung

Rekurrente neuronale Netzwerke

Long-Short-Term-Memory-(LSTM-)Netzwerke

Gated Recurrent Units (GRUs)

BiLSTM-Netzwerke

Einbettungen

Torchtext
Ein Twitter-Datensatz
Field-Objekte definieren
Einen Wortschatz aufbauen
Erstellung unseres Modells
Die Trainingsschleife modifizieren
Tweets klassifizieren
Datenaugmentation
Zufälliges Einfügen
Zufälliges Löschen
Zufälliges Austauschen
Rückübersetzung
Datenaugmentation und Torchtext
Transfer Learning?
Zusammenfassung
Weiterführende Literatur
6 Eine Reise in die Welt der Klänge
Töne
Der ESC-50-Datensatz
Den Datensatz beschaffen
Audiowiedergabe in Jupyter
Den ESC-50-Datensatz erkunden
SoX und LibROSA
torchaudio
Einrichten eines eigenen ESC-50-Datensatzes
Ein CNN-Modell für den ESC-50-Datensatz
Frequenzbereich
Mel-Spektrogramme

Ein neuer Datensatz
Ein vortrainiertes ResNet-Modell
Lernrate finden
Datenaugmentation für Audiodaten
Transformationen mit torchaudio
SoX-Effektketten
SpecAugment
Weitere Experimente
Zusammenfassung
Weiterführende Literatur
7 PyTorch-Modelle debuggen
3 Uhr morgens. Wie steht es um Ihre Daten?
TensorBoard
TensorBoard installieren
Daten an TensorBoard übermitteln
Hooks in PyTorch
Mittelwert und Standardabweichung visualisieren
Class Activation Mapping
Flammendiagramme
py-spy installieren
Flammendiagramme interpretieren
Eine langsame Transformation beheben
Debuggen von GPU-Problemen
Die GPU überwachen
Gradient-Checkpointing
Zusammenfassung
Weiterführende Literatur
8 PyTorch im Produktiveinsatz

Bereitstellen eines Modells
Einrichten eines Flask-Webdiensts
Modellparameter laden
Erstellen eines Docker-Containers
Unterschiede zwischen lokalem und Cloud-Speicher
Logging und Telemetrie
Deployment mit Kubernetes
Einrichten der Google Kubernetes Engine
Aufsetzen eines Kubernetes-Clusters
Dienste skalieren
Aktualisierungen und Bereinigungen
TorchScript
Tracing
Scripting
Einschränkungen in TorchScript
Mit libTorch arbeiten
libTorch einrichten
Ein TorchScript-Modell importieren
Quantisierung
Dynamische Quantisierung
Weitere Quantisierungsmöglichkeiten
Lohnt sich das alles?
Zusammenfassung
Weiterführende Literatur
9 Praxiserprobte PyTorch-Modelle in Aktion
Datenaugmentation: Vermischen und Glätten
Mixup
Label-Glättung

Computer, einmal in scharf bitte!
- Einführung in die Super-Resolution
- Einführung in Generative Adversarial Networks (GANs)
- Der Fälscher und sein Kritiker
- Trainieren eines GAN
- Die Gefahr des Mode Collapse
- ESRGAN

Weitere Einblicke in die Bilderkennung
- Objekterkennung
- Faster R-CNN und Mask R-CNN

Adversarial Samples
- Black-Box-Angriffe
- Abwehr adversarialer Angriffe

Die Transformer-Architektur
- Aufmerksamkeitsmechanismus
- Attention Is All You Need
- BERT
- FastBERT
- GPT-2
- GPT-2 vorbereiten
- Texte mit GPT-2 erzeugen
- Beispielhafte Ausgabe
- ULMFiT
- Welches Modell verwenden?

Selbstüberwachtes Training mit PyTorch Lightning auf Basis von Bildern
- Rekonstruieren und Erweitern der Eingabe
- Daten automatisch labeln
- PyTorch Lightning

Der Imagenette-Datensatz
Einen selbstüberwachten Datensatz erstellen
Ein Modell mit PyTorch Lightning erstellen
Weitere Möglichkeiten zur Selbstüberwachung (und darüber hinaus)
Zusammenfassung
Weiterführende Literatur
Index
Über den Autor
Über den Übersetzer

KAPITEL 4

Transfer Learning und andere Kniffe

Nachdem Sie sich die Architekturen im vorherigen Kapitel angesehen haben, fragen Sie sich vielleicht, ob Sie ein bereits trainiertes Modell herunterladen und noch weiter trainieren können. Die Antwort lautet: Ja! Dies stellt eine unglaublich effektive Technik im Deep Learning dar und wird als *Transfer Learning* bezeichnet. Dabei wird ein für eine Aufgabe (vor-)trainiertes Netzwerk (z.B. ImageNet) auf eine andere übertragen und in Anwendung gebracht (Fisch versus Katze).

Warum sollten Sie diesen Ansatz wählen? Wie sich zeigt, weiß eine auf ImageNet trainierte Architektur bereits sehr viel über Bilder, insbesondere auch ziemlich viel darüber, ob etwas einer Katze oder einem Fisch (oder einem Hund oder einem Wal) gleicht. Da Sie nicht mehr von einem im Wesentlichen blanken neuronalen Netzwerk ausgehen, werden Sie mit dem Ansatz des Transfer Learning wahrscheinlich bedeutend weniger Zeit in den Trainingsprozess investieren müssen, und darüber hinaus können Sie mit einem bedeutend kleineren Trainingsdatensatz auskommen. Herkömmliche Deep-Learning-Ansätze benötigen riesige Datenmengen, um gute Ergebnisse zu erzielen. Beim Transfer Learning können Sie Klassifikatoren auf Basis von ein paar Hundert Bildern erstellen, deren Leistungsvermögen dem eines Menschen gleicht.

Transfer Learning mit ResNet

Es liegt nahe, ein ResNet-Modell wie das in Kapitel 3 zu erstellen und es einfach in unsere bestehende Trainingsschleife einzufügen. Genau das können Sie nun machen! Es gibt nichts Magisches an dem ResNet-Modell; es besteht aus den gleichen Bausteinen, die Sie bereits gesehen haben. Allerdings ist das Modell recht groß. Obwohl Sie mit Ihren Daten eine spürbare Verbesserung gegenüber einem ResNet-Basismodell feststellen werden, benötigen Sie eine gewisse Menge an Daten, um zu gewährleisten, dass das *Trainingssignal* alle Teile der Architektur erreicht und sie maßgeblich auf Ihre neue Klassifizierungsaufgabe

hin trainiert. Bei diesem Ansatz versuchen wir, zu vermeiden, auf eine große Datenmenge zurückgreifen zu müssen.

Aber die Sache ist die: Wir haben es hier nicht mit einer Architektur zu tun, die mit zufälligen Parametern initialisiert wurde, wie wir es zuvor getan haben. Unser vortrainiertes ResNet-Modell enthält bereits eine Menge an Informationen, die für die Bilderkennungs- und Klassifizierungsbedürfnisse codiert sind, also warum sich die Mühe machen, es umzutrainieren? Stattdessen unterziehen wir das Netzwerk einem *Finetuning* bzw. einer *Feinabstimmung*. Wir ändern die Architektur leicht, um am Ende einen neuen Netzwerkblock einzufügen, der die `Linear`-Schichten der 1.000 Kategorien ersetzt, die normalerweise für die ImageNet-Klassifizierung zuständig sind. Wir *frieren* alle vorhandenen ResNet-Schichten ein und aktualisieren beim Training nur die Parameter in den neuen Schichten, nehmen aber weiterhin die Aktivierungen aus unseren eingefrorenen Schichten. Dadurch lassen sich unsere neuen Schichten schnell trainieren, während die Informationen, die die vortrainierten Schichten bereits enthalten, erhalten bleiben.

Zuerst laden wir ein vortrainiertes ResNet-50-Modell:

```
from torchvision import models

transfer_model = models.resnet50(pretrained=True)
```

Als Nächstes müssen wir die Schichten einfrieren. Dies erreichen wir auf einfache Art und Weise: Wir verhindern, dass sich die Gradienten kumulieren, indem wir die Funktion `requires_grad()` verwenden. Diese müssten wir für jeden Parameter im Netzwerk aufrufen. Hier bietet PyTorch eine `parameters()`-Methode, mit der wir das recht komfortabel handhaben können:

```
for name, param in transfer_model.named_parameters():

    param.requires_grad = False
```

Sie sollten die BatchNorm-Schichten in einem Modell nicht einfrieren, da sie darauf trainiert wurden, den Mittelwert und die Standardabweichung des Datensatzes, auf dem das Modell ursprünglich trainiert wurde, zu approximieren und nicht des Datensatzes, mit dem Sie die Feinabstimmung vornehmen möchten. Ein Teil des *Signals* Ihrer Daten kann am Ende verloren gehen, da die `BatchNorm`-Schicht Ihre Eingabe *korrigiert*. Sie können sich die Modellstruktur ansehen und nur die Schichten einfrieren, die keine `BatchNorm`-Schichten sind, wie es hier der Fall ist:

```
for name, param in
transfer_model.named_parameters():

    if("bn" not in name):

        param.requires_grad = False
```

Dann müssen wir den letzten Klassifizierungsblock durch einen neuen ersetzen, den wir für die Bestimmung von Katzen oder Fischen trainieren werden. In diesem Beispiel ersetzen wir ihn durch ein paar `Linear`-Schichten, eine `ReLU`- und eine `Dropout`-Schicht. Sie könnten hier jedoch auch zusätzliche Konvolutionsschichten einfügen. Glücklicherweise speichert die PyTorch-Implementierung von ResNet den letzten Klassifizierungsblock als Instanzvariable `fc`, sodass wir diese nur durch unsere neue Struktur ersetzen müssen (andere in PyTorch enthaltene Modelle verwenden entweder `fc` oder `classifier`. Sie sollten die Definition im Quelltext überprüfen, wenn Sie dies mit einem anderen Modelltyp versuchen):

```
import torch.nn as nn

transfer_model.fc =
nn.Sequential(nn.Linear(transfer_model.fc.in_features,500),

                         nn.ReLU(), nn.Dropout(),
                         nn.Linear(500,2))
```

Im vorhergehenden Code nutzen wir die Variable `in_features`, die uns erlaubt, die Anzahl der Aktivierungen zu erfassen, die in eine Schicht eingehen (2.048 in diesem Fall). Sie können auch `out_features` verwenden, um die Anzahl

ausgehender Aktivierungen zu ermitteln. Dies sind praktische Funktionen, wenn Sie Netzwerke wie Bausteine zusammenfügen; stimmen die eingehenden Merkmale einer Schicht nicht mit den ausgehenden Merkmalen der vorherigen Schicht überein, erhalten Sie bei der Ausführung einen Fehler.

Schließlich kehren wir zu unserer Trainingsschleife zurück und trainieren das Modell wie gewohnt. Sie sollten bereits nach wenigen Epochen eine bedeutsame Verbesserung in der Genauigkeit sehen können.

Das Transfer Learning ist eine bemerkenswerte Technik zur Verbesserung der Genauigkeit Ihrer Deep-Learning-Anwendung. Wir können aber noch eine Reihe weiterer Tricks anwenden, um die Leistung unseres Modells zu steigern. Schauen wir uns einige davon an.

Die optimale Lernrate finden

Sie erinnern sich vielleicht noch an meine Aussage aus Kapitel 2, als ich im Rahmen des Trainings neuronaler Netze bei der Einführung des Konzepts der *Lernrate* darauf verwiesen habe, dass dies einer der wichtigsten Hyperparameter ist, den man ändern kann. Dabei kam ich auch darauf zu sprechen, was für ein Wert dafür genutzt werden sollte, wobei ich einen tendenziell kleinen Wert vorschlug und Ihnen nahelegte, mit verschiedenen Werten zu experimentieren. Nun … die schlechte Nachricht ist, dass viele Leute auf ähnliche Weise die optimale Lernrate für ihre Architektur, normalerweise mit einer Methode namens *Gittersuche* (engl. Grid Search), entdecken. Dabei suchen sie mühsam in einer Untermenge von möglichen Lernratenwerten und vergleichen die Ergebnisse anhand eines Validierungsdatensatzes. Dieses Vorgehen ist unglaublich zeitaufwendig, und obwohl viele das so praktizieren, gibt es wiederum andere, die sich auf praktische Erfahrungswerte verlassen. Zum Beispiel liegt beim Adam-Optimierer ein empirisch häufig zu beobachtender Wert der Lernrate bei 3e-4. Dieser Wert ist bekannt als Karpathys Konstante, nachdem Andrej Karpathy (derzeit Direktor für KI bei Tesla) im Jahr 2016 darüber getwittert hat (*https://oreil.ly/WLw3q*). Leider lesen die wenigsten seinen nächsten Tweet: »Ich wollte nur sichergehen, dass die Leute verstehen, dass dies ein Witz ist.« Das Komische ist, dass 3e-4 ein Wert ist, der tatsächlich oft gute Ergebnisse liefert. Es ist also ein Witz, der dennoch ein Fünkchen Wahrheit beinhaltet.

Zum einen haben wir eine langsame und umständliche Suche und zum anderen lediglich ein *Gefühl* dafür, was eine gute Lernrate ist, das auf unklaren und undurchsichtigen Kenntnissen fußt, die man aus der Arbeit an unzähligen

Architekturen gewonnen hat. Gibt es einen besseren Weg als diese beiden Extreme?

Zum Glück lautet die Antwort Ja – obwohl Sie überrascht sein werden, wie viele Leute diese bessere Methode nicht anwenden. Ein etwas nebulöser Artikel von Leslie Smith, einer Wissenschaftlerin am US Naval Research Laboratory, enthielt einen Ansatz zur Ermittlung einer geeigneten Lernrate.[1] Doch erst Jeremy Howard brachte die Technik in seinem fast.ai-Kurs in den Fokus, sodass sie sich in der Deep-Learning-Gemeinschaft allmählich durchzusetzen begann. Die Idee ist ganz einfach: Im Laufe einer Epoche beginnt man mit einer kleinen Lernrate und erhöht sie mit jedem Minibatch, was zum Schluss der Epoche zu einer relativ hohen Rate führt. Berechnen Sie den Verlust für jede Lernrate und wählen Sie dann anhand eines Diagramms die aus, die den größten Rückgang bewirkt. Sehen Sie sich beispielsweise die Grafik in Abbildung 4-1 an.

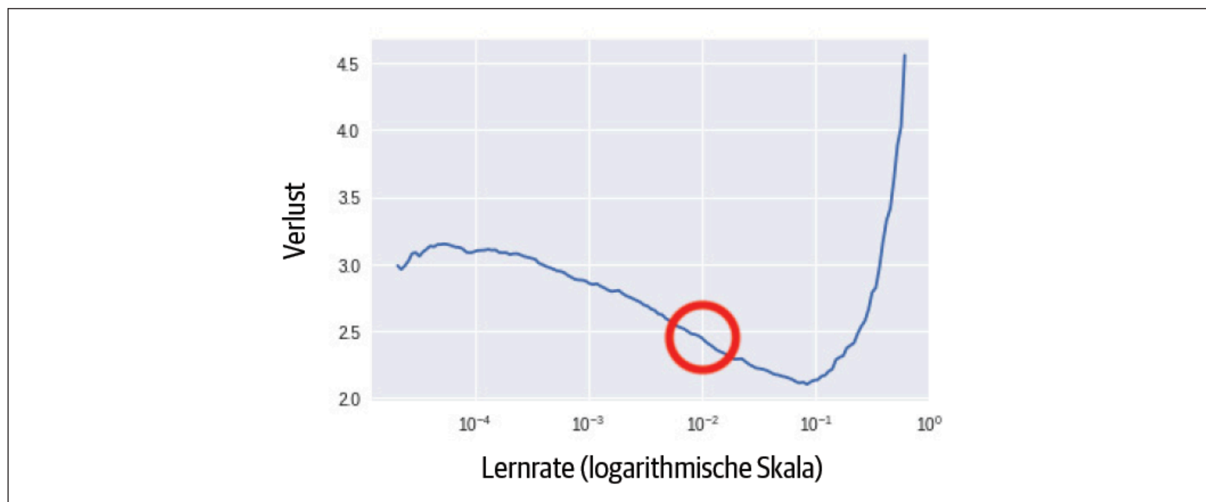


Abbildung 4-1: Der Zusammenhang zwischen Lernrate und Verlust

In diesem Fall sollten wir eine Lernrate von etwa 1e-2 (siehe Markierung) verwenden, da dies ungefähr der Punkt ist, an dem die Steigung des Abstiegs am steilsten ist.

Beachten Sie, dass Sie nicht nach dem niedrigsten Punkt der Kurve suchen, der vielleicht intuitiver erscheinen mag, sondern nach dem Punkt, an dem man am schnellsten zum unteren Ende der Kurve gelangt.

Hier ist eine vereinfachte Version dessen, was die fast.ai-Bibliothek im Hintergrund berechnet:

```
import math
```

```
def find_lr(model, loss_fn, optimizer, train_loader,

            init_value=1e-8, final_value=10.0, device="cpu"):

    number_in_epoch = len(train_loader) - 1

    update_step = (final_value / init_value) ** (1 /
    number_in_epoch)

    lr = init_value

    optimizer.param_groups[0]["lr"] = lr

    best_loss = 0.0

    batch_num = 0

    losses = []

    log_lrs = []

    for data in train_loader:

        batch_num += 1

        inputs, targets = data

        inputs = inputs.to(device)

        targets = targets.to(device)

        optimizer.zero_grad()
```

```python
outputs = model(inputs)

loss = loss_fn(outputs, targets)

# Abbruch bei explodierendem Verlust

if batch_num > 1 and loss > 4 * best_loss:

    if(len(log_lrs) > 20):

        return log_lrs[10:-5], losses[10:-5]

    else:

        return log_lrs, losses

# Speichern des kleinsten Verlustes

if loss < best_loss or batch_num == 1:

    best_loss = loss

# Speichern der Werte

losses.append(loss.item())

log_lrs.append((lr))

# Rückwärtsdurchlauf/Fehlerrückführung und Optimierung

loss.backward()

optimizer.step()
```

```
        # Aktualisierung der Variablen lr für den nächsten
        Schritt

        # und speichern

        lr *= update_step

        optimizer.param_groups[0]["lr"] = lr

    if(len(log_lrs) > 20):

        return log_lrs[10:-5], losses[10:-5]

    else:

        return log_lrs, losses

(lrs, losses) = find_lr(transfer_model,
torch.nn.CrossEntropyLoss(),

                         optimizer, train_data_loader,
                         device=device)
```

Wir iterieren über die Batches und trainieren das Netzwerk fast wie gewohnt; wir leiten unsere Eingaben durch das Modell und bekommen dann den resultierenden Verlust des jeweiligen Batches. Wir speichern den niedrigsten Verlust als best_loss ab und vergleichen diesen jeweils mit dem neuen Verlust. Wenn unser neuer Verlust mehr als das Vierfache des `best_loss` beträgt, brechen wir die Funktion ab und geben das zurück, was wir bis zu diesem Zeitpunkt als niedrigsten Verlust ermittelt haben (da der Verlust wahrscheinlich gegen unendlich strebt). Andernfalls fahren wir damit fort, den Verlust und den Wert der aktuellen Lernrate in die Liste aufzunehmen, und aktualisieren bzw. erhöhen die Lernrate schrittweise auf dem Weg zur maximalen Rate am Ende der Schleife.

Zuvor benötigen wir noch unsere zuvor definierten Objekte, die hier bequemerweise noch einmal aufgeführt sind:

```
import torch
import torch.optim as optim
import torchvision
from PIL import Image
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder
def check_image(path):
    try:
        im = Image.open(path)
        return True
    except:
        return False
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_path = "./chapter2/train"
train_data = ImageFolder(root=train_data_path,
                         transform=img_transforms,
                         is_valid_file=check_image)
val_data_path = "./chapter2/val/"
val_data = ImageFolder(root=val_data_path,
                       transform=img_transforms,
                       is_valid_file=check_image)
batch_size = 64
train_data_loader = DataLoader(train_data,
                         batch_size=batch_size,
                         shuffle=True)
val_data_loader = DataLoader(val_data,
                         batch_size=batch_size,
                         shuffle=True)
```

```
if torch.cuda.is_available():

    device = torch.device("cuda")

else:

    device = torch.device("cpu")

transfer_model.to(device)

optimizer = optim.Adam(transfer_model.parameters(), lr=0.001)

(lrs, losses) = find_lr(transfer_model,
torch.nn.CrossEntropyLoss(),

                       optimizer, train_data_loader,
                       device=device)
```

Der Verlust kann dann in Abhängigkeit von der logarithmierten Lernrate mit der matplotlib-Funktion plt veranschaulicht werden:

```
import matplotlib.pyplot as plt

plt.plot(lrs, losses)

plt.xscale("log")

plt.xlabel("log. Lernrate")

plt.ylabel("Verlust")

plt.show()
```

Beachten Sie, dass wir nur Teilausschnitte der Lernraten (lr) und Verluste zurückgeben, da uns die ersten und letzten Epochen des Trainingsvorgangs (besonders wenn man die Lernrate recht schnell erhöht) keine nennenswerten Informationen bieten.

Die Implementierung in der Bibliothek von fast.ai beinhaltet zusätzlich eine gewichtete Glättung, sodass Sie geglättete Linien in Ihrer Grafik erhalten, wohingegen dieser Codeausschnitt eine spitz verlaufende Ausgabe erzeugt. Denken Sie unbedingt daran, dass Sie Ihr Modell vorher speichern und neu laden sollten, um wieder in den Zustand gelangen zu können, in dem es sich vor dem Aufruf der Funktion `find_lr()` befand, da diese das Modell in der Tat trainiert und die Einstellungen der Lernrate des Optimierers verändert. Zudem sollten Sie den von Ihnen gewählten Optimierer reinitialisieren. Dies können Sie gleich vornehmen, indem Sie die Lernrate, die Sie beim Betrachten des Schaubilds ermittelt haben, einfach übernehmen!

Das beschert uns einen guten Wert für unsere Lernrate, aber mit *differenziellen Lernraten* können wir noch mehr rausholen.

Differenzielle Lernraten

Im bisherigen Trainingsprozess haben wir stets eine Lernrate auf das gesamte Modell angewandt. Wenn wir ein Modell von Grund auf neu trainieren, ergibt das wahrscheinlich Sinn, doch im Bereich des Transfer Learning können wir gewöhnlich eine leicht bessere Genauigkeit erzielen, wenn wir etwas anderes ausprobieren: Wir trainieren verschiedene Gruppen von Schichten mit unterschiedlichen Lernraten. Zu Beginn des Kapitels haben wir alle vortrainierten Schichten in unserem Modell eingefroren und nur unseren neuen Klassifikator trainiert. Doch vielleicht möchten wir einige der Schichten, z.B. des von uns verwendeten ResNet-Modells, feiner abstimmen und unser Modell durch das Training der Schichten, die unserem Klassifikator direkt vorausgehen, ein wenig akkurater werden lassen. Vielleicht benötigen sie im Vergleich zu unseren neuen Schichten nur wenig Training, da diese vorhergehenden Schichten bereits auf dem ImageNet-Datensatz trainiert wurden? PyTorch bietet eine einfache Umsetzungsmöglichkeit. Modifizieren wir zunächst unseren Optimierer für das ResNet-50-Modell:

```
import torch.optim as optim

found_lr = 1e-2
```

```
optimizer = optim.Adam([

    { 'params': transfer_model.layer4.parameters(), 'lr':
    found_lr /3},

    { 'params': transfer_model.layer3.parameters(), 'lr':
    found_lr /9},

    ], lr=found_lr)
```

Das setzt die Lernrate für layer4 (kurz vor unserem Klassifikator) auf ein Drittel der *ermittelten* Lernrate und auf ein Neuntel für `layer3`. Auch wenn diese Kombination in meinem Ansatz recht gut funktioniert hat, können Sie natürlich gern experimentieren. Es gibt allerdings noch eine weitere Sache. Erinnern Sie sich vielleicht an den Beginn dieses Kapitels? Wir haben alle diese vortrainierten Schichten *eingefroren*. Es ist gut und schön, ihnen eine unterschiedliche Lernrate zu geben, doch ab diesem Moment werden die Schichten in keiner Weise mehr durch den Trainingsvorgang beeinflusst, da keine Gradienten kumuliert werden. Das möchten wir ändern:

```
unfreeze_layers = [transfer_model.layer3,
transfer_model.layer4]

for layer in unfreeze_layers:

    for param in layer.parameters():

        param.requires_grad = True
```

Da die Parameter in diesen Schichten nun wieder Gradienten aufweisen, werden bei der Feinabstimmung des Modells die differenziellen Lernraten angewandt. Beachten Sie, dass Sie Teile des Modells nach Belieben einfrieren und auch wieder trainierbar machen und – sofern gewünscht – die weitere Feinabstimmung für jede Schicht separat vornehmen können!

Nachdem wir uns nun die Lernraten angesehen haben, untersuchen wir einen anderen Aspekt des Trainings unserer Modelle: die Daten, die wir in die Modelle einspeisen. Doch zuvor empfehle ich Ihnen noch, das Modell in gewohnter

Manier mit torch.save(model.state_dict(), catfishweights.pth) zu speichern, da wir in Kapitel 8 darauf zurückkommen möchten und die Modellparameterdatei für einen Webdienst benötigen.

Datenaugmentation

Eine der gefürchtetsten Aussagen in der Datenwissenschaft lautet: »Oh nein, mein Modell hat sich zu sehr an die Daten angepasst!« Wie ich in Kapitel 2 erwähnt habe, liegt eine *Überanpassung* dann vor, wenn das Modell dazu tendiert, die im Trainingsdatensatz dargestellten Daten einfach widerzuspiegeln, anstatt eine verallgemeinernde Lösung zu bieten. Häufig wird darüber geredet, wie ein bestimmtes Modell den Datensatz lediglich *auswendig gelernt* hat, was bedeutet, dass das Modell die entsprechenden Antworten memoriert hat, aber auf Daten im Produktiveinsatz nur eine unzureichende Leistung erzielt.

Die klassische Möglichkeit, dies zu vermeiden, besteht darin, große Datenmengen anzuhäufen. Durch die Betrachtung von mehr Daten erhält das Modell eine allgemeinere Vorstellung von der Aufgabe, die es zu lösen versucht. Betrachten wir die Situation als Komprimierungsproblem: Wenn Sie verhindern, dass das Modell einfach alle Antworten speichern kann (indem es seine Speicherkapazität mit so vielen Daten schlicht überfordert), ist es gezwungen, die Eingabe zu *komprimieren* und somit eine Lösung zu finden, bei der die Antworten nicht einfach innerhalb des Netzwerks abgespeichert werden können. Das ist durchaus sinnvoll und funktioniert auch gut. Was können wir jedoch tun, wenn wir nur 1.000 Bilder vorliegen haben und Transfer Learning anwenden möchten?

Ein Ansatz ist die sogenannte *Datenaugmentation* (engl. Data Augmentation). An einem Bild können wir eine Reihe von Maßnahmen anwenden, die eine Überanpassung verhindern und das Modell allgemeiner machen sollen. Betrachten Sie die Bilder von der Katze Helvetica in den Abbildungen 4-2 and 4-3.

Abbildung 4-2: Unser Ausgangsbild

Abbildung 4-3: Die umgedrehte Katze Helvetica

Für uns ist es offensichtlich, dass beide Bilder das Gleiche abbilden. Das zweite ist nur eine gespiegelte Kopie des ersten. Die Tensordarstellung wird anders sein, da die RGB-Werte nun an anderen Stellen des dreidimensionalen Bilds liegen. Aber es handelt sich immer noch um eine Katze, sodass das auf diesem Bild trainierte Modell hoffentlich lernen wird, eine Katzenform auf der linken oder rechten Seite des Bilds zu erkennen, anstatt einfach das gesamte Bild mit einer *Katze* zu assoziieren. Dies lässt sich in PyTorch einfach bewerkstelligen. Führen Sie sich noch einmal den Codeausschnitt aus Kapitel 2 (und diesem Kapitel) vor Augen:

```
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])
])
```

Dies ist unsere Transformationspipeline, die alle Bilder durchlaufen, bevor sie für das Training in das Modell gespeist werden. Die Bibliothek torchivision.transforms enthält jedoch noch viele andere Transformationsfunktionen, die zur Augmentation von Trainingsdaten verwendet werden können. Sehen wir uns einige der nützlicheren Funktionen an und vollziehen wir nach, wie sich einige der weniger offensichtlichen Transformationen auf Helvetica auswirken.

Transformationen in Torchvision

Das `torchvision`-Paket bietet eine große Auswahl potenzieller Transformationen, die zur Datenaugmentation verwendet werden können, sowie zwei Möglichkeiten, Transformationen selbst zu konzipieren. In diesem Abschnitt sehen wir uns die nützlichsten integrierten Transformationen sowie einige benutzerdefinierte Transformationen an, die Sie in Ihren eigenen Anwendungen nutzen können.

```
transforms.ColorJitter(brightness=0, contrast=0, saturation=0,
hue=0)
```

Die Funktion `ColorJitter` (Farbschwankung) ändert zufällig die Helligkeit, den Kontrast, die Sättigung und den Farbton eines Bilds. Für Helligkeit, Kontrast und Sättigung können Sie entweder eine Gleitkommazahl oder ein Tupel aus Gleitkommazahlen angeben, die nicht negativ sind und innerhalb des Bereichs zwischen 0 und 1 liegen. Die Zufälligkeit wird entweder zwischen 0 und der angegebenen Gleitkommazahl liegen, oder das Tupel wird verwendet, um einen Zufallseffekt innerhalb des angegebenen Gleitkommazahlenpaars zu erzeugen. Für den Farbton ist eine Gleitkommazahl oder ein Tupel mit Gleitkommazahlen zwischen –0,5 und 0,5 anzugeben, und es werden zufällige Farbtonanpassungen innerhalb des Bereichs [-*hue*, *hue*] bzw. [*min*, *max*] erzeugt (siehe Abbildung 4-4 als Beispiel).

Um Ihr Bild zu spiegeln, können Sie mit diesen beiden Transformationen ein Bild zufällig entlang der horizontalen bzw. vertikalen Achse spiegeln:

```
transforms.RandomHorizontalFlip(p=0.5)
```

```
transforms.RandomVerticalFlip(p=0.5)
```

Abbildung 4-4: Anwendung der ColorJitter-Transformation mit 0,5 als Wert für alle Parameter

Geben Sie für die Eintrittswahrscheinlichkeit der Spiegelung entweder eine Gleitkommazahl zwischen 0 und 1 an oder belassen Sie den voreingestellten Standardwert mit einer 50%igen Spiegelungswahrscheinlichkeit. In Abbildung 4-5 können Sie die Katze vertikal gespiegelt sehen.

Abbildung 4-5: Vertikale Spiegelung

Die `RandomGrayscale`-Funktion bietet eine ähnliche Transformation, nur dass sie hier die Graustufen des Bilds in Abhängigkeit vom Parameter *p* zufällig verändert (die Voreinstellung liegt bei 10%):

```
transforms.RandomGrayscale(p=0.1)
```

Mit RandomCrop und RandomResizeCrop schneiden Sie zufällig size große Ausschnitte des Bilds aus, wobei als size entweder eine Ganzzahl für Höhe und Breite oder ein Tupel für eine unterschiedliche Höhe und Breite angegeben werden kann. Abbildung 4-6 zeigt ein Beispiel mit der RandomCrop-Funktion in Aktion.

```
transforms.RandomCrop(size,
                        padding=None, pad_if_needed=False,
                        fill=0, padding_mode='constant')
transforms.RandomResizedCrop(size,
                        scale=(0.08, 1.0),
                        ratio=(0.75, 1.3333333333333333),
                        interpolation=2)
```

An dieser Stelle müssen Sie ein wenig aufpassen. Wenn Ihr Ausschnitt zu klein ist, laufen Sie Gefahr, nur unwichtige Teile des Bilds herauszuschneiden und das Modell infolgedessen mit den falschen Informationen zu trainieren. Angenommen, Sie hätten ein Bild, auf dem eine Katze auf einem Tisch spielt. Wenn Sie durch den Ausschnitt vor allem den Tisch (aber nicht die Katze) ausschneiden und dieser dann als *Katze* klassifiziert werden soll, hat das ungewollte Konsequenzen, da das Netzwerk ein falsches Verständnis davon bekommt, wie eine Katze aussieht (und auch ein Tisch). Während die RandomResizedCrop-Funktion die Größe des Ausschnitts so anpasst, dass er die vorgegebene Größe ausfüllt, kann RandomCrop einen Ausschnitt nahe am Rand und auch über das Bild hinaus auswählen.

Die Funktion `RandomResizedCrop` verwendet die bilineare Interpolation, aber Sie können auch den nächsten Nachbarn oder die bikubische Interpolation wählen, indem Sie den Parameter `interpolation` anpassen. Weitere Einzelheiten finden Sie in der PIL-Dokumentation (*https://oreil.ly/rNOtN*).

Wie Sie in Kapitel 3 gesehen haben, können wir das Bild auffüllen, um die gewünschte Größe des Bilds beizubehalten. Standardmäßig ist die Funktion auf `constant`-Padding eingestellt, das die ansonsten leeren Pixel außerhalb des Bilds mit dem für `fill` angegebenen Wert ausfüllt. Ich empfehle Ihnen jedoch, stattdessen das `reflect`-Padding zu verwenden, da es empirisch gesehen etwas besser zu funktionieren scheint, als einfach nur einen leeren, gleichmäßigen Bereich einzufügen.

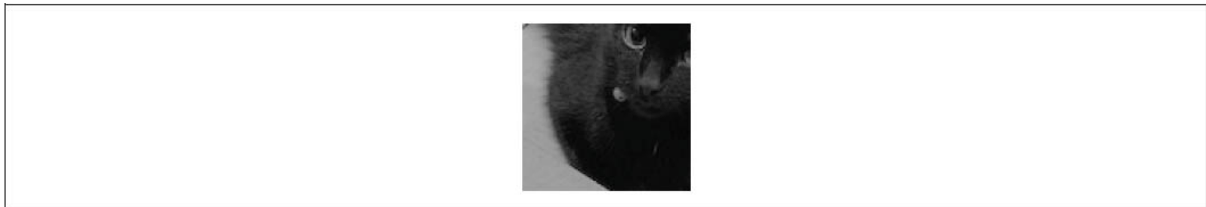


Abbildung 4-6: Die RandomCrop-Transformation mit size = 100

Wenn Sie ein Bild nach dem Zufallsprinzip drehen möchten, variiert die Drehung beim Einsatz der `RandomRotation`-Funktion zwischen `[-degrees, degrees]`, sofern `degrees` eine einzelne Gleitkomma- oder Ganzzahl ist, oder zwischen `(min,max)`, wenn es sich um ein Tupel handelt:

```
transforms.RandomRotation(degrees, resample=False,
expand=False, center=None)
```

Wenn `expand` auf `True` gesetzt ist, wird die Funktion das Ausgabebild so erweitern, dass es die komplette Rotation beinhaltet. Ansonsten ist sie standardmäßig so eingestellt, dass sie die Eingabe nur unter Ausnutzung ihrer Dimensionen beschneidet. Sie können einen PIL-Resampling-Filter angeben und optional ein `(x,y)`-Tupel für das Rotationszentrum bereitstellen. Andernfalls wird die Transformation um die Mitte des Bilds erfolgen. Abbildung 4-7 zeigt eine `RandomRotation`-Transformation, bei der `degrees` auf 45 Grad gesetzt ist.

Abbildung 4-7: Die RandomRotation-Transformation mit degrees = 45

Die Funktion `Pad` bietet eine vielseitige Transformationsmöglichkeit für das Padding, mit der die Ränder eines Bilds aufgefüllt werden können (zusätzliche Höhe und Breite):

```
transforms.Pad(padding, fill=0, padding_mode=constant)
```

Durch die Angabe eines einzelnen Werts in `padding` wird die Auffüllung hinsichtlich dieser Länge in alle Richtungen vorgenommen. Ein Zweier-Tupel für `padding` bewirkt eine Auffüllung hinsichtlich der Länge von (links/rechts, oben/unten), und ein `padding`-Wert mit einem Vierer-Tupel erzeugt eine Auffüllung nach dem Schema (links, oben, rechts, unten). In der Voreinstellung ist das Padding auf den Modus `constant` eingestellt. Dadurch wird der Wert von `fill` in die entsprechenden aufzufüllenden Felder kopiert. Andere Optionen sind `edge`, das die letzten Werte am Bildrand in die vorgegebene Auffüllungsfläche übernimmt, `reflect`, das die Werte des Bilds (mit Ausnahme des Rands) in den Padding-Bereich spiegelt, und `symmetric`, das ähnlich wie `reflect` wirkt, aber den letzten Wert des Bilds am Rand einschließt. Abbildung 4-8 zeigt die Auswirkungen, wenn wir `padding` auf 25 und `padding_mode` auf `reflect` setzen. Man kann gut erkennen, dass sich der Karton an den Rändern wiederholt.

Abbildung 4-8: Die Pad-Transformation mit padding = 25 und padding_mode = reflect

`RandomAffine` erlaubt Ihnen, zufällige affine Transformationen des Bilds zu spezifizieren – Skalierung, Rotationen, Verschiebungen (engl. Translations) und/oder Scherung (engl. Shearing) oder eine beliebige Kombination. Abbildung 4-9 zeigt ein Beispiel für eine affine Transformation.

```
transforms.RandomAffine(degrees,
                      translate=None, scale=None,
                      shear=None, resample=False, fillcolor=0)
```

Abbildung 4-9: Die RandomAffine-Transformation mit degrees = 10 und shear = 50

Der `degrees`-Parameter ist entweder eine einzelne Gleitkomma- oder Ganzzahl oder ein Tupel. Wenn Sie nur eine einzelne Zahl angeben, wird eine zufällige

Drehung zwischen (-*degrees*, *degrees*) erzeugt, mit einem Tupel eine zufällige Drehung zwischen (-*min*, *max*). Der `degrees`-Parameter muss explizit gesetzt werden, um Rotationen zu verhindern – es gibt keine Standardeinstellung. Der Parameter `translate` ist ein Tupel bestehend aus zwei Multiplikatoren, einem horizontalen und einem vertikalen Multiplikator (*horizontal_multipler*, *vertical_multiplier*). Zum Zeitpunkt der Transformation wird eine horizontale Verschiebung, `dx`, aus dem Bereich *–image_width × horizontal_multiplier < dx < img_width × horizontal_width* (auf Deutsch *–Bildbreite × horizontaler Multiplikator < dx < Bildbreite × horizontale Breite*) gesampelt, und eine vertikale Verschiebung wird auf die gleiche Weise in Bezug auf die Bildhöhe und den vertikalen Multiplikator stichprobenartig gezogen.

Die Skalierung (`scale`) wird durch ein weiteres Tupel angegeben (*min*, *max*), aus dem ein einheitlicher Skalierungsfaktor nach dem Zufallsprinzip ausgewählt wird. Die Scherung (`shear`) kann entweder eine einzelne Gleitkomma-/Ganzzahl oder ein Tupel sein, und die Zufallsstichproben werden auf die gleiche Weise wie beim `degrees`-Parameter gezogen. Der `resample`-Parameter ermöglicht die optionale Bereitstellung eines PIL-Resampling-Filters. Zu guter Letzt gibt `fillcolor` ebenfalls optional in Form einer Ganzzahl die Füllfarbe für Bereiche innerhalb des resultierenden Bilds an, die außerhalb der endgültigen Transformation liegen.

Was die Transformationen betrifft, die Sie in einer Datenaugmentationspipeline verwenden sollten, empfehle ich zu Beginn auf jeden Fall, die verschiedenen zufälligen Spiegelungen, Farbschwankungen, Rotationen und Ausschnitte zu verwenden.

Sie können noch weitere Transformationen im `torchvision`-Paket finden; nähere Einzelheiten hierzu finden Sie in der Dokumentation (*https://oreil.ly/b0Q0A*). Natürlich kann es auch vorkommen, dass Sie selbst eine Transformation für Ihre spezielle Datendomäne erstellen möchten, die nicht standardmäßig enthalten ist. Wie Sie noch im weiteren Verlauf des Kapitels sehen werden, bietet PyTorch verschiedene Möglichkeiten, benutzerdefinierte Transformationen zu implementieren.

Farbräume und Lambda-Transformationen

Es mag vielleicht etwas trivial erscheinen, dies überhaupt zur Sprache zu bringen, aber bisher haben wir unsere gesamte Bildbearbeitung im eher standardmäßigen 24-Bit-RGB-Farbraum durchgeführt, in dem jedes Pixel einen Rot-, einen Grün- und einen Blauwert hat (8 Bit pro Kanal), die die Farbe dieses Pixels codieren. Es sind jedoch auch andere Farbräume verfügbar!

Eine beliebte Alternative ist der HSV-Farbraum, bei dem drei 8-Bit-Werte für den *Farbton*, die *Sättigung* und den *Wert* vorhanden sind. Es gibt die Meinung, dass dieses System das menschliche Sehvermögen besser nachbildet als der traditionelle RGB-Farbraum. Aber warum ist das wichtig? Ein Gebirge im RGB-Farbraum ist doch auch ein Gebirge im HSV-Farbraum, oder?

Nun, die neuere Deep-Learning-Forschung zu diesem Thema zeigt einige Belege dafür, dass andere Farbräume eine etwas höhere Genauigkeit als der RGB-Farbraum erzeugen können. Ein Berg mag ein Berg sein, aber der Tensor, der sich in der Darstellung jedes Farbraums ergibt, wird unterschiedlich sein. So kann ein Farbraum manche Dinge in Ihren Daten besser erfassen als ein anderer.

In Kombination mit Ensembles könnten Sie leicht eine Reihe von Modellen erstellen, die die Ergebnisse des Trainings in den RGB-, HSV-, YUV- und LAB-Farbräumen kombinieren, um Ihrer Vorhersagepipeline ein paar Prozentpunkte mehr Genauigkeit abzugewinnen.

Ein gewisses Problem besteht darin, dass PyTorch keine Transformation bietet, die das bewerkstelligen könnte. Aber es offeriert eine Reihe von Werkzeugen, mit denen wir ein Bild nach dem Zufallsprinzip vom Standard-RGB- in den HSV-Farbraum (oder in einen anderen Farbraum) umwandeln können. Wenn wir in die PIL-Dokumentation schauen, sehen wir, dass die Funktion `Image.convert()` zur Verfügung steht, um ein PIL-Bild von einem Farbraum in einen anderen zu übersetzen. Wir könnten für diese Konvertierung eine benutzerdefinierte Transformationsklasse schreiben. PyTorch bietet uns jedoch eine Klasse `transforms.Lambda` an, mit der wir problemlos jede Funktion einbinden und der Transformationspipeline zur Verfügung stellen können. Hier ist unsere benutzerdefinierte Funktion:

```
def _random_colour_space(x):

    output = x.convert("HSV")

    return output
```

Diese wird dann in eine Klasse `transforms.Lambda` eingebunden und kann in alle gewöhnlichen Transformationspipelines – wie solche, die wir bereits betrachtet haben – eingebunden werden:

```
colour_transform = transforms.Lambda(lambda x:
_random_colour_space(x))
```

Das genügt zwar, wenn wir *jedes* Bild in HSV umwandeln wollten, aber das möchten wir eigentlich nicht. Wir möchten stattdessen, dass die Bilder jedes Batches nach dem Zufallsprinzip geändert werden, sodass es wahrscheinlich ist, dass das Bild in verschiedenen Epochen in unterschiedlichen Farbräumen abgebildet wird. Wir könnten unsere ursprüngliche Funktion aktualisieren, um eine Zufallszahl zu erzeugen, und diese verwenden, um eine Zufallswahrscheinlichkeit für die Änderung des Bilds zu erzeugen. Stattdessen betreiben wir aber noch weniger Aufwand und nutzen die Funktion `RandomApply`:

```
random_colour_transform =
transforms.RandomApply([colour_transform])
```

Standardmäßig besitzt `RandomApply` einen Parameter `p` mit einem Wert von `0.5`, sodass eine 50:50-Chance besteht, dass die Transformation angewendet wird. Experimentieren Sie mit dem Hinzufügen weiterer Farbräume und der Wahrscheinlichkeit der Anwendung der Transformation, um zu sehen, welchen Effekt dies auf unsere Katze-oder-Fisch-Klassifizierung hat.

Schauen wir uns eine weitere benutzerdefinierte Transformation an, die allerdings etwas komplizierter ist.

Benutzerdefinierte Transformationsklassen

Manchmal reicht eine einfache Lambda-Funktion nicht aus; vielleicht möchten Sie zum Beispiel eine Initialisierung oder einen Zustand im Auge behalten. In diesen Fällen können Sie eine benutzerdefinierte Transformation erstellen, die entweder mit PIL-Bilddaten oder einem Tensor arbeitet. Eine solche Klasse muss zwei Methoden implementieren: `__call__`, die die Transformationspipeline während des Transformationsprozesses aufruft, und `__repr__`, die eine Zeichenkettendarstellung der Transformation zusammen mit jedem Zustand zurückgeben sollte, der für Diagnosezwecke nützlich sein kann.

Im folgenden Code implementieren wir eine Transformationsklasse, die einem Tensor zufälliges gaußsches Rauschen hinzufügt. Wenn die Klasse initialisiert wird, übergeben wir den Mittelwert und die Standardabweichung des von uns gewünschten Rauschens, und während der `__call__`-Methode ziehen wir eine Stichprobe aus dieser Verteilung und fügen sie dem eingehenden Tensor hinzu:

```
class Noise():

    """Fügt dem Tensor gaußsches Rauschen hinzu.

    >>> transforms.Compose([

    >>>     transforms.ToTensor(),

    >>>     Noise(0.1, 0.05)),

    >>> ])

"""

def __init__(self, mean, stddev):

    self.mean = mean

    self.stddev = stddev

def __call__(self, tensor):

    noise = torch.zeros_like(tensor).normal_(self.mean,
    self.stddev)

    return tensor.add_(noise)

def __repr__(self):

    repr = (

            f"{self.__class__.__name__ }"
```

```
            f"(mean={self.mean}, stddev={self.stddev})"

    )
```

Wenn wir die Klasse in eine Pipeline einbinden, können wir die Ausgabe der `__repr__`-Methode sehen, die dadurch aufgerufen wird:

```
transforms.Compose([Noise(0.1, 0.05)])

>> Compose(

    Noise(mean=0.1, sttdev=0.05)

)
```

Da die Klasse `transforms` keinen Einschränkungen unterliegt und einfach von der Python-Basisobjektklasse erbt, können Sie alles Denkbare tun. Möchten Sie zum Zeitpunkt der Programmausführung ein Bild komplett durch eines aus der Google-Bildsuche ersetzen? Das Bild durch ein völlig anderes neuronales Netz laufen lassen und das Ergebnis in die Pipeline weiterleiten? Wenden Sie eine Reihe von Bildtransformationen an, die das Bild in einen verrückten, reflektierenden Schatten seines ursprünglichen Selbst verwandeln? Alles möglich, wenn auch nicht wirklich empfehlenswert – obwohl es interessant zu sehen wäre, ob der *Twirl*-Transformationseffekt von Photoshop die Genauigkeit schlechter oder besser macht! Warum probieren Sie es nicht einfach aus?

Abgesehen von Transformationen gibt es noch ein paar mehr Möglichkeiten, aus einem Modell so viel Leistung wie möglich herauszuholen. Sehen wir uns weitere Beispiele an.

Klein anfangen und schrittweise vergrößern!

Hier ist ein Tipp, der zwar etwas eigenartig erscheint, aber tatsächlich zu besseren Resultaten führt: Fangen Sie klein an und gehen Sie nach und nach zu Größerem über. Damit meine ich, dass, wenn Sie ein Modell mit 256 × 256 Bildern trainieren möchten, Sie ein paar weitere Datensätze erstellen sollten, bei denen die Bilder auf 64 × 64 und 128 × 128 Pixel skaliert sind. Erstellen Sie Ihr Modell mit dem 64 × 64-Datensatz, nehmen Sie die Feinabstimmung wie üblich

vor und trainieren Sie dann *dasselbe Modell* mit dem 128 × 128-Datensatz – nicht von Grund auf, sondern mit den bereits trainierten Parametern. Sobald es so aussieht, als hätten Sie das meiste aus den 128 × 128 Daten herausgekitzelt, gehen Sie zu Ihrem eigentlichen 256 × 256-Datensatz über. Sie werden wahrscheinlich ein oder zwei Prozentpunkte Verbesserung hinsichtlich der Genauigkeit feststellen.

Wir wissen zwar nicht genau, warum das funktioniert, aber eine Theorie besagt, dass das Modell durch das Training bei den niedrigeren Auflösungen die Gesamtstruktur des Bilds kennenlernt und dieses Wissen verfeinern kann, wenn die eingehenden Bilder eine höhere Auflösung besitzen. Doch das ist nur eine Theorie. Das ändert jedoch nichts daran, dass wir damit ein Ass im Ärmel haben, wenn wir aus einem Modell das letzte Quäntchen Leistung herausholen möchten.

Wenn Sie nicht möchten, dass mehrere Kopien eines Datensatzes gleichzeitig den Speicher belegen, können Sie die Skalierung auch on-the-fly mit der Funktion `Resize` der `torchvision`-Transformationen umsetzen:

```
resize = transforms.Compose([ transforms.Resize((64,64)),

            …_andere Datenaugmentationstransformationen_…

            transforms.ToTensor(),

            transforms.Normalize([0.485, 0.456, 0.406],

                        [0.229, 0.224, 0.225])

])
```

Der Nachteil dabei ist, dass Sie am Ende mehr Zeit für das Training aufwenden müssen, da PyTorch die Skalierung immer wieder durchführen muss. Wenn Sie vorher alle Bilder in ihrer Größe ändern, würden Sie den Trainingsvorgang wahrscheinlich verkürzen, was allerdings auf Kosten des Speicherplatzes auf Ihrer Festplatte geht. Aber sind wir nicht immer mit diesem Kompromiss konfrontiert?

Das Konzept, klein anzufangen und dann zu Größerem überzugehen, lässt sich auch auf Architekturen übertragen. Zunächst eine ResNet-Architektur wie ResNet-18 oder ResNet-34 zu verwenden, um verschiedene Transformationsansätze zu erproben und ein Gefühl dafür zu bekommen, wie das Training funktioniert, bietet eine viel schnellere Feedback-Schleife, als wenn man mit einem ResNet-101- oder ResNet-152-Modell beginnt. Fangen Sie klein an und vergrößern Sie die Architektur Schritt für Schritt. Die kleineren Modelle können Sie durchaus zur Vorhersage wiederverwenden, indem Sie sie in ein Ensemble-Modell integrieren.

Ensemble-Modelle

Was ist besser als ein Modell, das Vorhersagen trifft? Nun, wie wäre es mit einer ganzen Reihe von Vorhersagen? *Ensembling* ist eine Technik, die in traditionelleren Methoden des maschinellen Lernens ziemlich verbreitet ist und auch im Bereich des Deep Learning recht gut funktioniert. Die Idee hierbei ist, mehrere Vorhersagen aus einer Vielzahl von Modellen zu erhalten und diese Vorhersagen miteinander zu kombinieren, um eine abschließende Antwort zu erhalten. Da verschiedene Modelle in verschiedenen Bereichen unterschiedliche Stärken haben, wird eine Kombination all ihrer Vorhersagen hoffentlich zu einem genaueren Ergebnis führen als ein Modell allein.

Es gibt zahlreiche Ansätze für Ensembles, und wir werden hier nicht auf alle eingehen. Stattdessen widmen wir uns einer einfachen Möglichkeit, mit Ensembles zu beginnen, eine, die meiner Erfahrung nach noch einmal 1% mehr Genauigkeit bringt – mitteln Sie einfach die Vorhersagen:

```
# Unter der Annahme, dass Sie eine Liste von Modellen haben

# und Ihre Eingabe Ihr Eingabetensor ist.

predictions = [m.fit(input) for m in models]

avg_prediction = torch.stack(predictions).mean(0).argmax()
```

Mit der `stack`-Methode werden die Tensoren miteinander verknüpft. Wenn wir also an unserer Katze-oder-Fisch-Aufgabe arbeiten würden und vier Modelle in unserem Ensemble hätten, würden wir am Ende einen 4 × 2-Tensor erhalten, der aus vier 1 × 2-Tensoren gebildet wird. Die `mean`-Funktion bildet den Mittelwert,

wobei wir als Dimensionsangabe eine 0 übergeben müssen, um sicherzustellen, dass der Mittelwert über die erste Dimension gebildet wird, anstatt dass einfach alle Tensorelemente addiert werden und eine skalare Ausgabe erzeugt wird. Schließlich wählt die Funktion `argmax` den Tensorindex mit dem höchsten Element aus, wie Sie es zuvor bereits gesehen haben.

Es ist ein Leichtes, sich noch komplexere Ansätze auszudenken. Vielleicht könnten die Vorhersagen jedes einzelnen Modells des Ensembles gewichtet und diese Gewichtungen entsprechend angepasst werden, wenn ein Modell eine richtige oder falsche Antwort liefert. Welche Modelle sollten Sie verwenden? Ich habe festgestellt, dass eine Kombination von ResNet-Architekturen (z.B. 34, 50, 101) recht gut funktioniert. – Nichts hindert Sie übrigens daran, Ihr Modell regelmäßig zu speichern und im Laufe der Zeit verschiedene Momentaufnahmen des Modells in Ihrem Ensemble zu verwenden!

Zusammenfassung

Wir kommen zum Ende von Kapitel 4 und lassen nun die Thematik der Bildverarbeitung hinter uns, um zur Verarbeitung von Textdaten überzugehen. Sicherlich verstehen Sie inzwischen, wie neuronale Konvolutionsnetze mit Bilddateien funktionieren, und Sie haben jetzt auch eine ganze Reihe von Tricks wie Transfer Learning, Lernratenermittlung, Datenaugmentation und Ensembling in petto, die Sie in Ihrer spezifischen Anwendungsdomäne zum Einsatz bringen können.

Weiterführende Literatur

Wenn Sie daran interessiert sind, mehr zum Thema Bildverarbeitung zu erfahren, empfehle ich Ihnen, sich den fast.ai (*https://fast.ai*)-Kurs von Jeremy Howard, Rachel Thomas und Sylvain Gugger anzuschauen. Wie ich bereits erwähnt hatte, haben wir in diesem Kapitel zur Ermittlung der Lernrate eine vereinfachte Version aus ihrem Kurs verwendet. Doch der Kurs geht auf viele der Methoden in diesem Kapitel noch detaillierter ein. Die auf PyTorch aufbauende fast.ai-Bibliothek bietet sich ebenfalls für Ihre Projekte an, da sie sich auf simple Art und Weise für Aufgaben im Bereich der Bild- und auch der Textverarbeitung einsetzen lässt.

- Torchvision-Dokumentation (*https://oreil.ly/vNnST*)
- PIL/Pillow-Dokumentation (*https://oreil.ly/Jlisb*)

- »Cyclical Learning Rates for Training Neural Networks« (*https://arxiv.org/abs/1506.01186*) von Leslie N. Smith (2015)
- »ColorNet: Investigating the Importance of Color Spaces for Image Classification« (*https://arxiv.org/abs/1902.00267*) von Shreyank N. Gowda und Chun Yuan (2019)

Index

Symbole

@app.route()-Funktion 158

@torch.jit.script_method 173

A

AdaGrad 29

Adam, Optimierer 29, 109

AdaptiveAvgPool-Schicht 44

AdaptiveMaxPool-Schicht 44

add_graph()-Funktion 133

Adversarial Samples 205–209

  und Abwehr adversarialer Angriffe 208

  und Black-Box-Angriffe 208

AlexNet 46, 151, 170

Amazon Web Services *siehe* AWS

AMD 2

Anaconda 10

Angriffe

  adversariale 208

  Black Box 208

  White Box 208

Apache MXNet XV

append_effect_to_chain 121

ARG 161

argmax()-Funktion 12, 35, 93, 206

arXiv 236

arXiv Sanity Preserver 236

Audio *siehe* Töne

Aufmerksamkeitsmechanismus 209

Austauschen, zufälliges 95
Autoencoder 195
AutoML 50
AWS (Amazon Web Services) 5, 166
Azure 5, 6, 166
Azure Blob Storage 163, 164
Azure Marketplace 7

B

Backpropagation Through Time 79
backward()-Funktion 30
BadRandom 147, 190
BatchNorm-Schicht 53, 58, 200
batch_size, Batchgröße 23
BCEWithLogitsLoss()-Funktion 92
benutzerdefinierte Transformationsklassen 72
BertLearner.from_pretrained_model 213
best_loss 62
Bidirectional Encoder Representations from Transformers (BERT) 211–214, 225
Bilderkennung 200–205
  Faster R-CNN und Mask R-CNN zur 203–205
  Objekterkennung für 201–203
Bildklassifizierung 17–37
  Aktivierungsfunktionen 25
  Beispiel 17
  Daten zur 19
  Erstellen eines Netzwerks 25
  Erstellen eines Validierungs- und Testdatensatzes 22
  Herausforderungen bei 18–24
  Netzwerk trainieren für 29
  neuronale Netzwerke 24–29
  Optimierung 27–29
  Speichern von Modellen 35
  Trainingsdatensatz erstellen zur 21–22
  und DataLoader 20
  und GPU 31

Verlustfunktionen 26
Vorhersagen 34
biLSTM (bidirectional LSTM) 81
Bitcoin 2
Black-Box-Angriffe 208
BookCorpus-Datensatz 211
Borg 166
Broadcasting, Tensor 14
C
C++ Compiler 176
C++-Bibliothek *siehe* libTorch
Cache, LRU, least recently used (am wenigsten kürzlich benutzter) 115
Caffe2 185
__call__ 72, 124
CAM (Class Activation Mapping) 137–140
Canadian Institute for Advanced Research (CIFAR-10) 129
Chainer XIII
checkpoint_sequential 152
CIFAR-10 (Canadian Institute for Advanced Research), Datensatz 129, 205
Class Activation Mapping (CAM) 129, 137–140
classifier 59
Cloud-Plattformen 3–8
Amazon Web Services 5
Anbieter 5–8
Azure 6
Google Cloud Platform 7
Wahl 8
Cloud-Speicher 164
lokaler versus Cloud-Speicher 163–165
CMake 176
CNNs *siehe* neuronale Konvolutionsnetze
Codebeispiele, Beschaffung und Anwendung XXII
Codierung, 1-aus-n 82
Colaboratory (Colab) 4
ColorJitter 66

Compute Unified Device Architecture (CUDA) 9, 149
conda 105, 156
Conv2d-Schicht 40–43, 194
COPY 161
copyfileobj() 164
CPU 2, 147
CrossEntropyLoss()-Funktion 26, 92, 109, 191, 192
CUDA (Compute Unified Device Architecture) 9, 149
cuda.is_available()-Funktion 11
cuda()-Funktion 31

D

Dataset-Klasse, Definition 20
Dateisystem 34
Daten
  Augmentation *siehe* Datenaugmentation
  Bildklassifizierung 19
  laden und konvertieren 20
  torchtext 85
  Trainingsdatensatz erstellen 21–22
  unausgewogene 104
  Validierungs- und Testdatensätze 22
Datenaugmentation 64–74, 94–98
  benutzerdefinierte Transformationsklassen 72
  Farbräume und Lambda-Transformationen 71
  klein anfangen mit 73
  Label-Glättung 192
  mit Audiodaten 120–127
  mixup 187–192
  Rückübersetzung 96
  torchtext 97
  Transfer Learning und 98
  Transformationen in Torchvision 66–71
  vermischen und glätten 187–193
  zufälliges Austauschen 95
  zufälliges Einfügen 94

zufälliges Löschen 95
Datensätze
Arten 23
für Frequenzbereich 113–117
Training 21–22
Validierungs-/Test- 22
WikiText-103 223
DDR4 3
Debugging 129–153
Flammendiagramme 140–148
GPU-Probleme 149–153
TensorBoard und 130–140
Decoder 195
Deep Learning, Definition XIII
degrees-Parameter 70
DenseNet 49
Destillation 208
differenzielle Lernraten 63
DigitalOcean 166
Diskriminatornetzwerke 197, 199
Docker 203
Docker Hub 160
Docker-Container, erstellen 160–162
download.py-Skript 19, 22
Dropout XII, 58
Dropout-Schicht 45, 46, 171

E

Einbettungen, für Textklassifizierung 82–84
Einbettungsmatrix 83
Einfügen, zufälliges 94
1-aus-n-Codierung 82
Encoder 195
Enhanced Super-Resolution Generative Adversarial Network (ESRGAN) 200
ENTRYPOINT 162
ENV 161

Environmental-Sound-Classification-(ESC-) Datensatz 102–109
Audiowiedergabe in Jupyter für 103
beschaffen 102
CNN-Modell für 108
erkunden 103
erstellen 106
SoX und LibROSA für 105
torchaudio 105
epsilon 207
ESC-50 *siehe* Environmental-Sound-Classification-(ESC-)Datensatz
explodierender Gradient 79
EXPOSE 162
F
Facebook XI, 130
Faltungsmatrix 41
Farbräume
Datenaugmentation mit 71
HSV 71
Fast Gradient Sign Method (FGSM) 206
fast.ai-Bibliothek 60, 224
FastBERT 212–214
Faster R-CNN 203–205
fc 59
Feature Map 41
Filter 41
find_lr()-Funktion 63, 118
fit()-Funktion 213
fit_one_cycle 224
Flammendiagramme 140–148
Beheben langsamer Transformation 145–148
interpretieren 143
und py-spy installieren 143
Flask 156–159
forward()-Funktion 25, 40, 144
Fourier-Transformation 12

Frequenzbereich 111–120
Datensätze für 113–117
Mel-Spektrogramme 111–113
und Frequenzmaskierung 123–125
und Lernrate 118
und ResNet 117

G

GANs *siehe* Generative Adversarial Networks
Gated Recurrent Units (GRUs) 80
gc.collect()-Funktion 151
GCP (Google Cloud Platform) 5, 7
GCP Marketplace 7
Generative Adversarial Networks (GANs) 196–200
neuronale Netzwerke 197
Training 197
und ESRGAN 200
und Mode Collapse 199
Generatornetzwerk 197
__getitem__ 106, 113
get_stopwords()-Funktion 94
get_synonyms()-Funktion 94
Gittersuche 59
GKE (Google Kubernetes Engine) 166
Google XIV
Google Cloud Platform (GCP) 5, 7
Google Cloud Storage 163, 164
Google Colaboratory 4
Google Kubernetes Engine (GKE) 166
Google Translate 77, 97
GoogLeNet 46
googletrans 96
GPT-2 209, 214–223, 225
GPU (Graphical Processing Unit)
Anstieg XII
Bildklassifizierung 31

CNNs und 45
Debugging von Problemen bei 149–153
Flammendiagramme 147
für maßgeschneiderten Deep-Learning-Rechner 2
Gradient-Checkpointing 151–153
Matrixmultiplikation 190
überwachen 149
Gradient
explodierender 79
verschwindender 79
Gradient-Checkpointing 151–153
Gregg, Brendan 140
GRUs (Gated Recurrent Units) 80
H
Heatmap 137
Hooks 134
Howard, Jeremy 60
HSV-Farbraum 71
I
Image.convert()-Funktion 71
ImageFolder 21
ImageNet XII, 19, 117
ImageNet Large Scale Visual Recognition Challenge XII
import torch 5
imsave 204
Inception 46, 113, 127, 203
in_channels 42
in_features 59
__init__ 121, 175
init()-Funktion 25
In-Place-Funktion 13
Internet 6
item()-Funktion 12, 93
J
JIT-(Just-in-Time-)Tracing-Engine 170

Joyent 140
Jupyter Notebook 8
  Abspielen von Audio des ESC-50 103
  in AWS 6
  in Azure 7
Just-in-Time-(JIT-)Tracing-Engine 170
K
k80 3, 6
k8s *siehe* Kubernetes
Kaggle 54, 203
Karpathy, Andrej 59
Karpathys Konstante 59
Keras XV
Kollisionen 191
Kontrollkästchen 7
Konvolutionen 40–43
Kubernetes (k8s) 160, 166–169
  Aktualisierungen und Bereinigungen 169
  Cluster aufsetzen 167
  Dienste skalieren 168
  Einrichten auf GKE 166
L
Label 129
label_from_df 225
Label-Glättung 192, 208
labeln, gelabelt 19, 88, 129, 208
Lambda-Transformationen 71
langsame Transformation, beheben 145–148
Learning
  Deep *siehe* Deep Learning
  Transfer *siehe* Transfer Learning
__len__ 106
LeNet-5 45–50
Lernen
  überwacht 19

unüberwacht 19
Lernrate
differenziell 63
festlegen 28
und Frequenzbereich 118
und ResNet 59–63
LibROSA 105, 111
libTorch 175–179
Importieren eines TorchScript-Modells 177
installieren und einrichten 176
Linear-Schicht 58
list_effects()-Funktion 122
load()-Funktion 106
load_model()-Funktion 159, 163
load_state_dict()-Funktion 144
Logging 165
log_spectogram.shape 112
lokaler Speicher, Cloud-Speicher versus 163–165
Long-Short-Term-Memory-(LSTM-)Netzwerke 79–80
bidirektional 81
Gated Recurrent Units (GRUs) 80
ULMFiT und 223
Löschen, zufälliges 95
LRU-Cache, am wenigsten kürzlich benutzter (least recently used) 115
LSTM-Netzwerke *siehe* Long-Short-Term-Memory-(LSTM-)Netzwerke
Lua XIII
M
MacBook 10
Mask R-CNN 203–205
maskrcnn-benchmark-Bibliothek 203
maßgeschneiderter Deep-Learning-Rechner
CPU/Motherboard 2
GPU 2
RAM 2
Speicher 3

matplotlib 116
matplotlib-Bibliothek 63
max()-Funktion 12
MaxPool2d-Schicht 44
MaxPool-Schicht 46
max_size-Parameter 89
max_width-Parameter 124
md5sum 10
mean-Funktion 75
Mel-Skala 111
Mel-Spektrogramme 111–113
Microsoft Azure *siehe* Azure
Microsoft Cognitive Toolkit 185
Mittelwert, visualisieren 135
Mixup 187–192
MNIST 129
MobileNet 49
Mode Collapse 199
model.children()-Funktion 135
model.eval()-Funktion 31, 171
model.train()-Funktion 30
Modell bereitstellen 155–166
  Docker-Container 160–162
  Einrichten der Modellparameter 159
  Einrichten eines Flask-Webdiensts 156–159
  Logging und Telemetrie 165
  und lokaler versus Cloud-Speicher 163–165
models.alexnet(pretrained=True)-Funktion 50
Motherboard 2
MSELoss 27
Multihead Attention 210
MXNet 185
MySQL 140
N
n1-standard-1-Knoten 167

NamedTemporaryFile()-Funktion 164
NASNet 50
Natural Language Processing (NLP) 77
NC6 6
NCv2 7
Netzwerk, erstellen 25
neuronale Konvolutionsnetze (CNNs) 39–55
AlexNet 46
Architekturen 45–50
Beispiel 39
Dropout 45
ESC-50-Modell 108
Geschichte XI
Inception/GoogLeNet 46
Konvolutionen 40–43
Pooling 43
ResNet 49
VGG 47
neuronale Netzwerke
Aktivierungsfunktionen 25
erstellen 25
Geschichte XI
Optimierung 27–29
rekurrente 77–79
Verlustfunktionen 26
zur Bildklassifizierung 24–29
NLP (Natural Language Processing) 77
nn.Module 192
nn.Sequential()-Funktion 40
nn.Sequential()-Schicht 135
nn.Sequential-Schicht 151, 194
NumPy 14
NVIDIA GeForce RTX 2080 Ti 2, 149
NVIDIA GTX 1080 Ti 2, 5
NVIDIA RTX 2080 Ti 2, 3

nvidia-smi 150

O

Objekterkennung 201–203

OK 157

ones()-Funktion 12

ONNX (Open Neural Network Exchange) 184

OpenAI 209, 214

optim.Adam()-Funktion 29

Optimierung neuronaler Netzwerke 27–29

optimizer.step()-Funktion 30

out_channels 42

out_features 59

P

P100 6

P2, P3 5

p2.xlarge 6

Padding-Token 90

pandas 85

parameters()-Funktion 58

partial()-Funktion 135

PCPartPicker 3

permute()-Funktion 14

pip 105, 156

plt-Funktion 63

Pod 167

Pooling in neuronalen Konvolutionsnetzen 43

predict()-Funktion 158

predictions 204

preprocess()-Funktion 93

print()-Funktion 135, 171

print(model)-Funktion 54

process()-Funktion 93

Produktionsumgebung, Deployen einer PyTorch-Anwendung in 155–185

 Bereitstellen eines Modells 155–166

 Deployen mit Kubernetes 166–169

Docker-Container 160–162
Einrichten der Modellparameter 159
Einrichten eines Flask-Webdiensts 156–159
libTorch 175–179
Logging und Telemetrie 165
lokaler versus Cloud-Speicher 163–165
TorchScript 169–175
Programmaufruf 161, 163
py-spy 143, 146
Python 135, 155
Python 2.x 9
PyTorch (allgemein) 1–15
Cloud-Plattformen und 3–8
Installation 9–11
Tensoren und 11–15
Ursprung XIII
Zusammenbau eines maßgeschneiderten Deep-Learning-Rechners 1–3
PyTorch Hub 54

R

Raina, Rajat XII
RAM 2
RandomAffine 70
RandomApply 72
RandomCrop 67
RandomGrayscale 67
RandomResizeCrop 67
README 103
Rectified Linear Unit *siehe* ReLU
Red Hat Enterprise Linux (RHEL) 7 10
register_backward_hook()-Funktion 135
rekurrente neuronale Netzwerke (RNNs) 77–79, 209
ReLU (Rectified Linear Unit) 25, 37, 46, 58
remove()-Funktion 135
__repr__ 72
requires_grad() 58

resample 70
reshape()-Funktion 13
Resize(64,64)-Transformation 22
ResNet-152 149
ResNet-18 134
ResNet-Architektur 49, 54, 113, 203
  Transfer Learning mit 57–59
  und Frequenzbereich 117
  und Lernrate 59–63
RHEL (Red Hat Enterprise Linux) 7 10
RMSProp 29
RNNs (Recurrent Neural Networks) 77–79, 209
ROCm 2
Rückübersetzung 96
RUN 161

S

Salesforce XIV
save()-Funktion 106
savefig 116
Schichten
  AdaptiveAvgPool-Schicht 44
  AdaptiveMaxPool-Schicht 44
  BatchNorm-Schicht 53, 58, 200
  Conv2d-Schicht 40–43, 194
  Dropout-Schicht 45, 46, 171
  Linear-Schicht 58
  MaxPool2d-Schicht 44
  MaxPool-Schicht 46
  nn.Sequential-Schicht 135, 151, 194
  torch.nn.ConvTranspose2d-Schicht 194
  Upsample-Schicht 195
Scripting 172
Secure Shell (SSH) 5
Segmentierung 202
send_to_log()-Funktion 165

Sentiment140-Datensatz 85
seq2seq 78
SimpleNet 37
simplenet.parameters()-Funktion 29
Skalierung 70, 168
Smith, Leslie 60
softmax()-Funktion 27
Softmax-Aktivierungsfunktion 25
SoX 105
sox_build_flow_effects() 122
SoX-Effektketten 121
spaCy 85
SpecAugment 122–127
  Frequenzmaskierung 123–125
  Zeitmaskierung 125–127
Speicher
  in maßgeschneidertem Deep-Learning-Rechner 3
  lokal versus Cloud 163–165
Speichern von Modellen 35
SqueezeNet 49
SSH (Secure Shell) 5
Stacktrace 141
Standardabweichung, visualisieren 135
state_dict()-Funktion 159
stochastischer Gradientenabstieg (SGD) 29
SummaryWriter 131
Super-Resolution 193–200
  Beispiel 194–196
  ESGRAN 200
  und GANs 196–200
  und GANs trainieren 197
  und Generator- und Diskriminatornetzwerk 197
  und Mode Collapse 199

T

Telemetrie 165

tensor.mean()-Funktion 124

TensorBoard 130–140

Class Activation Mapping 137–140

Daten übermitteln an 131–133

installieren 130

Mittelwert und Standardabweichung visualisieren mit 135

und PyTorch-Hooks 134

Tensoren 12–14

Broadcasting 14

Operationen 12–14

TensorFlow XIV, 169

Tensorprozessoren (TPUs) XII, 7

TeslaV100 3, 6

Testdatensätze, erstellen 22

Textgenerierung, mit GPT-2 215–223

Textklassifizierung 77–99

biLSTM 81

Datenaugmentation 94–98

Einbettungen für 82–84

Gated Recurrent Units (GRUs) 80

in LSTMs 79–80

rekurrente neuronale Netzwerke (RNNs) 77–79

Rückübersetzung 96

torchtext 84–94

und Transfer Learning 98

zufälliges Austauschen 95

zufälliges Einfügen 94

zufälliges Löschen 95

tf.keras XV

Theano XV

to()-Funktion 13, 31

Töne 101–128

Datenaugmentation mit Audiodaten 120–127

Entstehung 101

Frequenzbereich 111–120

Frequenzmaskierung 123–125
in Jupyter Notebook 103
Mel-Spektrogramme 111–113
SoX-Effektketten 121
SpecAugment 122–127
torchaudio 105
Transformationen mit torchaudio 120
und ESC-50-Datensatz 102–109
Zeitmaskierung 125–127
Top-5 46, 49
torch.argmax()-Funktion 158
torch.distribution.Beta 190
torch.eq()-Funktion 33
torch.hub.list(pytorch/vision)-Funktion 55
torch.jit.save 172, 173
torch.jit.save()-Funktion 175, 177
torch.load()-Funktion 35
torch.max()-Funktion 33
torch.nn.ConvTranspose2d-Schicht 194
torch.save()-Funktion 35, 159
torch.topk()-Funktion 139
torch.utils.checkpoint_sequential()-Funktion 151
torch.utils.tensorboard 131
torchaudio 105, 120
torchaudio.load() 122
torchaudio.sox_effects.effect_names()-Funktion 122
torchaudio.sox_effects.SoxEffectsChain 121
TorchScript 169–175
Einschränkungen 174–175
libTorch 177
Scripting 172
Tracing 170–172
torchtext 84–94
Aktualisieren der Trainingsschleife 92
Daten für 85

- Field-Objekte definieren 87–89
- Modell erstellen 91
- und Datenaugmentation 97
- Vorhersagen mit 93
- Wortschatz aufbauen für 89–91

torchtext.datasets 85
torchvision 21
torchvision.models 55
TPUs (Tensor Processing Units) XII, 7
Tracing 170–172
train()-Funktion 34
train_net.py 205
Transfer Learning 57–75
- benutzerdefinierte Transformationsklassen 72
- Ensembling 74
- Farbräume und Lambda-Transformationen 71
- klein anfangen mit 73
- mit ResNet 57–59
- Transformationen in Torchvision 66–71
- und Datenaugmentation 64–74
- und differenzielle Lernraten 63
- und U-Net-Architektur 203

Transformationen
- Beheben langsamer 145–148
- in Torchvision 66–71
- mit torchaudio 120

Transformer-Architektur 209–225
- Aufmerksamkeitsmechanismus 209
- auswählen 225
- BERT 211–214
- FastBERT 212–214
- GPT-2 214–223
- Multihead Attention 210
- ULMFiT 223–225

transforms.ToTensor()-Funktion 21

TWEET 88, 93
Twitter 85
U
Uber XIV
Überanpassung (Overfitting) 22, 64
überwachtes Lernen 19
Ubuntu 10
ULMFiT 223–225
Umformen eines Tensors 13
unbekanntes Wort-Token 90
U-Net-Architektur 202
unsqueeze()-Funktion 35, 158
unüberwachtes Lernen 19
Upsample-Schicht 195
urlopen()-Funktion 164
V
Validierungsdatensätze 22
Verlustfunktionen 26
verschwindender Gradient 79
view()-Funktion 13, 26
Visdom 130
Visual Geometry Group (VGG) 47, 127
Vorhersagen
 mit torchtext 93
 und Ensembling 74
 zur Bildklassifizierung 34
vortrainierte Modelle 50–54
 auswählen 54
 BatchNorm 53
 Struktur eines Modells untersuchen 51–53
W
Waitress (Webserver) 162
Wellenform 101, 111, 120
White-Box-Angriff 208
WikiText-103-Datensatz 223

Word2vec 84

X

XLNet 236

Z

Z370 2

Zeitpunkt 78

Zeitschritt 78

zeroes()-Funktion 12

zero_grad()-Funktion 30

zufälliges Austauschen 95

zufälliges Einfügen 94

zufälliges Löschen 95