

# Einführung in maschinelles Lernen mit Python und PyTorch

## Semesterbegleitendes Einführungsskript - Ziffern erkennen

Rahmen: deutsche Hochschule, 14 Wochen, 2 SWS Vorlesung und 2 SWS Übung. Vorausgesetzt werden Python-Grundlagen (Variablen, Schleifen, Funktionen und Klassen). Ableitungen, Vektoren und Matrizen werden anwendungsbezogen wiederholt. Stand: 6. Oktober 2026.

Dieses Skript ist ein kompakter Einstieg in überwachtes maschinelles Lernen mit neuronalen Netzen. Es deckt nicht alle Teilgebiete der KI ab. Suche, Wissensrepräsentation, Sprachmodelle und Reinforcement Learning werden zur Einordnung erwähnt, aber nicht praktisch vertieft.

## Leitidee und Lernziele

Wir entwickeln einen „Ziffernassistenten“: Ein  $28 \times 28$  Pixel großes Graustufenbild wird einer Ziffer von 0 bis 9 zugeordnet. Das Beispiel bleibt über das ganze Semester gleich. Zunächst untersuchen wir die Daten und eine einfache Vergleichslösung. Danach entwickeln wir ein lineares Modell, ein mehrschichtiges Netz und ein kleines Faltungsnetz. Abschließend prüfen wir eigene Bilder und dokumentieren Grenzen.

Nach dem Semester können Studierende eine Klassifikationsaufgabe formulieren, Tensorformen erklären, einen Datenfluss implementieren, Gradienten und Training nachvollziehen, geeignete Metriken berechnen sowie ein Modell speichern und für neue Eingaben benutzen. Sie können außerdem Datenleckage, Überanpassung und irreführende Aussagen über Modellwahrscheinlichkeiten erkennen.

Die Übung ist kein Abschreiben der vollständigen Referenz. Studierende führen pro Woche eine eigene Datei `w01.py` bis `w14.py` beziehungsweise ein eigenes Notebook. Die Datei `kurs.py` ist eine ausführbare Referenz für den späteren Gesamtstand. Jeder Versuch erhält ein eigenes Ausgabeverzeichnis. Zum Protokoll gehören Fragestellung, Änderung, Messwert und Interpretation.

## Semesterübersicht

| Woche | Vorlesung                                    | Praktischer Meilenstein                |
|-------|----------------------------------------------|----------------------------------------|
| 1     | Lernaufgabe, Arbeitsumgebung, Baseline       | Umgebung prüfen, Baseline formulieren  |
| 2     | Tensoren, Dimensionen und Bildrepräsentation | Datenformen durch Assertions prüfen    |
| 3     | Datensätze, Batches und Datentrennung        | Reproduzierbaren Datenfluss aufbauen   |
| 4     | Lineare Algebra und lineare Klassifikation   | Erstes Modell mit zehn Ausgaben        |
| 5     | Loss, Ableitungen und Autograd               | Gradienten analytisch überprüfen       |
| 6     | Optimierung und Trainingsschleife            | Parameter durch Lernen verändern       |
| 7     | Evaluation und Fehlermetriken                | Konfusionsmatrix interpretieren        |
| 8     | Mehrschichtige neuronale Netze               | Lineares Modell und MLP vergleichen    |
| 9     | Generalisierung und Regularisierung          | Überanpassung kontrolliert untersuchen |
| 10    | Faltung und Pooling                          | Kleines CNN implementieren             |

| Woche | Vorlesung                                  | Praktischer Meilenstein                 |
|-------|--------------------------------------------|-----------------------------------------|
| 11    | CPU, CUDA und Laufzeitmessung              | Rechenzeit fair vergleichen             |
| 12    | Speicherung und Inferenz                   | Vorhersagen nach Neustart reproduzieren |
| 13    | Eigene Daten, Robustheit und Verantwortung | Verteilungswechsel untersuchen          |
| 14    | Abschlussversuch und Verteidigung          | Ergebnis nachvollziehbar präsentieren   |

## Organisation und Arbeitsaufwand

Vorschlag je Vorlesung: 10 Minuten Rückblick, 25 Minuten Begriffe und Herleitung, 20 Minuten Beispiel, 25 Minuten gemeinsame Codeanalyse, 10 Minuten Verständnisfragen. Je Übung: 10 Minuten Startkontrolle, 55 Minuten Pflichtaufgaben, 15 Minuten Auswertung, 10 Minuten Sicherung. Zusätzlich etwa zwei bis drei Stunden Selbststudium pro Woche; eine ECTS-Zuordnung muss zum Modulhandbuch passen.

Die hier vorgeschlagene Prüfungsform ist ein Projektbericht mit kurzer mündlicher Erläuterung. Sie ersetzt keine geltende Prüfungsordnung. Vorschlag für die Bewertung: fachliche Erklärung 25 %, korrekte Implementierung 25 %, nachvollziehbare Experimente 25 %, Fehleranalyse 15 %, Dokumentation 10 %. GPU-Besitz und besonders hohe Accuracy dürfen keinen eigenen Bewertungsvorteil darstellen.

## Vorbereitung des Rechnerpools

Alle Aufgaben sind auf CPU bearbeitbar. Installieren Sie ein zueinander passendes Paar aus PyTorch und torchvision mit dem offiziellen Installationsselektor. Für CUDA hängt der Befehl von Betriebssystem und unterstützter Compute-Plattform ab. Die von `nvidia-smi` angezeigte CUDA-Version ist kein Nachweis, dass das Python-Paket CUDA nutzen kann. Maßgeblich sind unter anderem Treiber, Paket und `torch.cuda.is_available()`.

Linux/macOS:

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
```

Windows PowerShell:

```
py -m venv .venv
.\.venv\Scripts\Activate.ps1
python -m pip install --upgrade pip
```

Anschließend den passenden Installationsbefehl von <https://pytorch.org/get-started/locally/> ausführen. Optional für Notebook-Arbeit: `python -m pip install jupyterlab matplotlib NumPy` wird im eigenen Kurscode nicht direkt benötigt; Bibliotheken können es als Abhängigkeit installieren. Nach erfolgreicher Einrichtung sichern Sie die konkrete Paketliste mit `python -m pip freeze > environment.txt`.

```
python kurs.py --smoke
python kurs.py --model linear --epochs 1 --limit 2000 --out run_probe
```

Der erste Aufruf benötigt keinen Datendownload. Der zweite lädt beim ersten Mal MNIST. Die Lehrperson stellt den Ordner `data/MNIST` vor der ersten Datensatzübung bereit, wenn Rechner keinen Internetzugang haben. Zufallsbilder eignen sich zur Prüfung von Formen und Rechenwegen, aber nicht als Ersatz für einen Lernversuch mit echten Ziffern. Für diese Unterlagen wurden Code-Syntax und Dokumentstruktur geprüft; ein Trainingslauf wurde in der Erstellungsumgebung

mangels PyTorch nicht ausgeführt. Vor Kursbeginn ist der obige Probelauf auf dem Rechnerpool durchzuführen.

# 1. Was bedeutet „lernen“?

## Konzept

Bei klassischer Programmierung beschreiben wir die Entscheidungsregel ausdrücklich. Für Handschrift wäre eine Regel wie „eine geschlossene Schleife bedeutet 0“ unzuverlässig: Auch 6, 8 und 9 können Schleifen enthalten. Beim überwachten Lernen geben wir Beispiele  $x$  und richtige Klassen  $y$  vor. Ein parametrisiertes Modell  $f_{\theta}$  erzeugt aus  $x$  eine Ausgabe. Lernen bedeutet,  $\theta$  anhand der Trainingsdaten so zu verändern, dass eine definierte Verlustfunktion kleiner wird.

Ein Modell lernt statistische Zusammenhänge. Es versteht die Zahl nicht automatisch im menschlichen Sinn. Die zentrale Frage lautet daher nicht nur „Wie gut merkt es sich Beispiele?“, sondern „Wie gut funktioniert es auf bisher ungesehenen, passenden Eingaben?“. Training und Inferenz sind verschiedene Phasen: Beim Training verändern wir Parameter; bei der Inferenz verwenden wir die erlernten Parameter für eine Vorhersage.

## Durchgängiges Beispiel

Eingabe ist ein Bild, Ausgabe eine der zehn Klassen. Eine triviale Baseline sagt immer die häufigste Klasse der Trainingsdaten voraus. Das ist ein Mindestvergleich, noch kein lernendes neuronales Netz. Bei exakt gleich häufigen Klassen erreicht sie 10 % Accuracy. Auf einem konkreten Datensatz muss der tatsächliche Wert gemessen werden.

```
import torch
print('PyTorch:', torch.__version__)
print('CUDA verfügbar:', torch.cuda.is_available())
print(torch.rand(2, 3))
```

## Übung 1 - 90 Minuten

1. (15 min) Prüfen Sie Python, virtuelle Umgebung und PyTorch. Dokumentieren Sie Version und verfügbares Gerät.
2. (25 min) Formulieren Sie Eingabe, Zielklasse, Trainingsphase und Inferenzphase des Ziffernassistenten mit eigenen Worten.
3. (30 min) Schreiben Sie eine Funktion `accuracy(y_true, y_pred)` für Python-Listen. Für `[0,1,2,2]` und `[0,2,2,1]` muss sie 0,5 liefern. Legen Sie fest, wie leere Listen und unterschiedliche Längen behandelt werden.
4. (20 min) Diskutieren Sie, warum 100 % Treffer auf vier selbst gewählten Bildern keine belastbare Qualitätsaussage sind.

Abgabe und Kontrolle: `w01.py` mit drei Testfällen und ein kurzer Steckbrief der Lernaufgabe. Die Funktion muss Eingaben prüfen und korrekt durch die Anzahl der Beispiele teilen. Zusatz: Ermitteln Sie die Baseline für die Klassenliste `[0,0,1,2,0]`.

## Verständnisfrage

Wird bei jeder Vorhersage automatisch weitergelernt? Nein. Im hier entwickelten System ändern sich Parameter nur in der ausdrücklich ausgeführten Trainingsschleife.

## 2. Tensoren: Zahlen erhalten Bedeutung durch ihre Verwendung

### Konzept

Ein Tensor ist in dieser Einführung ein mehrdimensional angeordnetes Zahlenfeld. Ein Skalar hat keine Achse, ein Vektor eine Achse, eine Matrix zwei. `torch.rand(5)` hat Form  $(5,)$ , `torch.rand(1,5)` Form  $(1,5)$ . Beide enthalten fünf Zahlen, unterscheiden sich aber bei Indexierung, Broadcasting und vielen Schnittstellen. Die erste Achse der zweiten Form kann beispielsweise für ein einzelnes Beispiel in einem Batch stehen.

Ein Tensor weiß nicht von sich aus, dass er ein Bild darstellt. Diese Bedeutung kommt aus Datensatz und Modellkonvention. Für Bildnetze verwenden wir NCHW: Anzahl der Beispiele, Kanäle, Höhe, Breite. Unser MNIST-Batch hat Form  $(N, 1, 28, 28)$ . Die Zielklassen sind dagegen ein ganzzahliger Tensor der Form  $(N,)$ ; das sind Klassenindizes, keine zufälligen Wahrscheinlichkeitsvektoren.

### Beispiel und Prüfungen

```
import torch
x = torch.rand(4, 1, 28, 28)
y = torch.tensor([2, 7, 0, 9], dtype=torch.long)
assert x.shape == (4, 1, 28, 28)
assert y.shape == (4,)
flat = x.flatten(start_dim=1)
assert flat.shape == (4, 784)
single = x[0]           # (1,28,28)
batch_of_one = x[0:1]   # (1,1,28,28)
```

`flatten(start_dim=1)` bewahrt die Batchachse. `flatten()` ohne dieses Argument würde alle Beispiele aneinanderhängen. Broadcasting erweitert kompatible Achsen rechnerisch; dadurch kann auch ein unbeabsichtigter Ausdruck gültig sein. Prüfen Sie Formen deshalb ausdrücklich, bevor Sie Loss oder Metriken berechnen.

### Übung 2

1. (15 min) Erzeugen Sie Skalar, Vektor, Zeilenmatrix und Bildbatch. Notieren Sie `shape`, `ndim`, `dtype` und `numel()`.
2. (25 min) Erklären Sie den Unterschied zwischen `x[0]` und `x[0:1]`. Stellen Sie eine verlorene Batchachse mit `unsqueeze(0)` wieder her.
3. (30 min) Erzeugen Sie zwei kleine Matrizen. Vergleichen Sie elementweise Multiplikation `*` und Matrixmultiplikation `@` anhand der Werte und Formen.
4. (20 min) Prüfen Sie, warum  $(4,)$  und  $(4,1)$  bei Subtraktion zu  $(4,4)$  führen können. Entwickeln Sie eine Assertion, die diesen Fehler bei einem Vektorvergleich verhindert.

Kontrolle: Flatten ergibt 784 Merkmale pro Bild; vier Bilder dürfen nicht zu einem Vektor mit 3136 Merkmalen verschmelzen. Zusatz: Berechnen Sie den reinen Speicherbedarf von 64 Bildern mit float32:  $64 \times 1 \times 28 \times 28 \times 4 = 200704$  Byte, ohne Verwaltungsdaten.

## 3. Daten laden und sauber trennen

### Konzept

Ein Dataset liefert ein einzelnes Paar aus Eingabe und Ziel. Ein DataLoader bündelt solche Paare zu Batches. `ToTensor()` wandelt bei den MNIST-PIL-Bildern die Pixel in Fließkommazahlen im Bereich 0 bis 1 um und ergänzt die Kanalachse. Eine Batchgröße von 64 bedeutet, dass gewöhnlich 64 Beispiele gemeinsam verarbeitet werden. Der letzte Batch kann kleiner sein.

Wir teilen die 60000 Beispiele des offiziellen Trainingsbestands in 55000 Trainings- und 5000 Validierungsbeispiele. Die 10000 offiziellen Testbeispiele bleiben bis nach der Modellwahl unberührt. Training bestimmt Parameter; Validierung hilft bei Architektur, Lernrate und Epochenzahl; der abschließende Test schätzt die Leistung der festgelegten Lösung. Wer wiederholt auf den Testdaten optimiert, macht sie faktisch zu Validierungsdaten.

### Beispiel

```
from torchvision import datasets, transforms
from torch.utils.data import random_split, DataLoader
import torch
full = datasets.MNIST('data', train=True, download=True,
                     transform=transforms.ToTensor())
train, valid = random_split(full, [55000, 5000],
                           generator=torch.Generator().manual_seed(2026))
loader = DataLoader(train, batch_size=64, shuffle=True, num_workers=0)
x, y = next(iter(loader))
assert x.shape[1:] == (1, 28, 28)
assert y.dtype == torch.long
assert 0 <= x.min() and x.max() <= 1
assert set(train.indices).isdisjoint(valid.indices)
```

Die feste Split-Zufallsquelle ist unabhängig vom Trainingsseed. So bleibt der Vergleich zwischen Versuchen auf denselben Daten möglich. Ein Seed erleichtert Wiederholbarkeit, garantiert aber keine identischen Resultate auf jeder Hardware und in jeder Softwareversion.

### Übung 3

1. (15 min) Laden Sie MNIST und betrachten Sie sechs Bilder mit ihren Zielklassen, optional mit matplotlib.
2. (25 min) Implementieren Sie den Split und prüfen Sie Größen sowie Überschneidungsfreiheit.
3. (30 min) Zählen Sie die Klassenhäufigkeiten ausschließlich im Training. Bestimmen Sie die häufigste Klasse und messen Sie deren Baseline auf der Validierung.
4. (20 min) Vergleichen Sie Batches mit und ohne `shuffle`. Erklären Sie, warum Validierung keine zufällige Reihenfolge benötigt.

Kontrolle: Jede Ausgangsposition gehört genau einer Teilmenge an. Baseline wird aus Trainingshäufigkeiten festgelegt. Zusatz: Berechnen Sie die Anzahl der Trainingsbatches für Batchgröße 64 ohne `drop_last`: 860.

## 4. Lineares Modell: vom Pixel zu zehn Scores

### Konzept

Ein Bild wird zu einem Vektor mit 784 Komponenten. Ein lineares Modell berechnet für jede der zehn Klassen einen Score:  $z_k = \sum_j W_{kj} \cdot x_j + b_k$ .  $W$  hat in `nn.Linear` Form  $(10, 784)$ ,  $b$  Form  $(10,)$ . Für einen Batch  $X$  ist die Rechnung  $X @ W.T + b$ ; die Ausgabe hat Form  $(N, 10)$ .

Die Scores heißen Logits und dürfen negativ oder größer als eins sein. Für die Klassenentscheidung wählen wir die größte Komponente. Softmax wandelt die Logits in positive Werte um, deren Summe eins ist. Diese Modellwahrscheinlichkeiten sind jedoch nicht automatisch kalibrierte Zuverlässigkeitswerte. Ein lineares Modell kann nur begrenzte Entscheidungsgrenzen im ursprünglichen Merkmalsraum darstellen.

```
import torch
from torch import nn
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))
x = torch.rand(4, 1, 28, 28)
logits = model(x)
predicted = logits.argmax(dim=1)
assert logits.shape == (4, 10)
assert predicted.shape == (4,)
```

### Übung 4

1. (20 min) Rechnen Sie ein Beispiel mit zwei Merkmalen und drei Klassen von Hand durch. Verwenden Sie  $X = [1, 2]$ , Gewichtszeilen  $[1, 0]$ ,  $[0, 1]$ ,  $[1, 1]$  und Bias null.
2. (25 min) Implementieren Sie dieselbe Rechnung mit Tensoren und vergleichen Sie sie mit `nn.Linear(2, 3)` nach gezieltem Setzen der Parameter unter `torch.no_grad()`.
3. (25 min) Implementieren Sie das MNIST-Modell und prüfen Sie die Ausgabeform für  $N = 1, 7$  und  $64$ .
4. (20 min) Zählen Sie Parameter manuell und mit `sum(p.numel() for p in model.parameters())`.

Kontrolle: Handscores sind  $[1, 2, 3]$ ; Ziffernmodell hat  $784 \times 10 + 10 = 7850$  Parameter. Ein untrainiertes Modell ist nur eine zufällige Startregel. Zusatz: Warum kann eine lineare Abbildung allein keine XOR-Klassifikation lösen?

# 5. Verlustfunktion und automatische Ableitung

## Konzept

Accuracy zählt richtige Entscheidungen, vermittelt aber kaum Information darüber, wie ein Score verändert werden sollte. Die Kreuzentropie berücksichtigt die Wahrscheinlichkeit der richtigen Klasse:  $L = -\log(p_y)$ . Bei  $p_y = 0,5$  beträgt der Verlust ungefähr 0,693; bei  $p_y = 0,9$  ungefähr 0,105. Für einen Batch wird standardmäßig der Mittelwert berechnet.

CrossEntropyLoss erwartet hier Logits der Form  $(N, 10)$  und Zielindizes  $(N, )$  vom Typ `long`. Eine zusätzliche Softmax vor dieser Loss-Funktion ist im Kursmodell falsch, weil die Funktion die passende LogSoftmax-Rechnung bereits enthält. Für die Anzeige von Wahrscheinlichkeiten bei der Inferenz ist Softmax dagegen sinnvoll.

Autograd speichert die für Ableitungen benötigten Operationen. Für  $L(w) = (w \cdot x - y)^2$  gilt  $dL/dw = 2(w \cdot x - y)x$ . Gradienten zeigen die Richtung des stärksten lokalen Anstiegs. Ein Gradientenabstieg bewegt den Parameter in die Gegenrichtung.

```
import torch
w = torch.tensor(1.0, requires_grad=True)
x, target = 2.0, 5.0
loss = (w*x-target)**2
loss.backward()
assert w.grad.item() == -12.0
```

## Übung 5

1. (20 min) Berechnen Sie Loss und Gradient des skalaren Beispiels von Hand.
2. (25 min) Prüfen Sie Autograd mit  $w = 1$  und  $w = 3$ . Vergleichen Sie zusätzlich eine zentrale numerische Ableitung mit kleinem  $h$ .
3. (25 min) Berechnen Sie Kreuzentropie für einen Batch des Ziffernmodells. Prüfen Sie, dass der Loss endlich und skalar ist.
4. (20 min) Untersuchen Sie, was bei zweimaligem Rückwärtsrechnen auf neu aufgebauten Graphen ohne Zurücksetzen von `w.grad` passiert.

Kontrolle: Bei  $w = 1$  sind Loss 9 und Gradient  $-12$ . Bei  $w = 3$  sind Loss 1 und Gradient 4. Gradienten akkumulieren. Zusatz: Leiten Sie mit Kettenregel den Gradienten eines Neurons mit mehreren Eingängen her.

## 6. Training: vier Schritte, viele mögliche Fehler

### Konzept

Für jeden Batch berechnen wir Vorhersage und Loss, löschen alte Gradienten, führen Backpropagation aus und verändern Parameter mit dem Optimierer. Die Lernrate bestimmt die Schrittweite. Eine zu kleine Lernrate kann Training verlangsamen; eine zu große kann das Ziel überspringen oder instabil werden. Eine Epoche ist ein Durchlauf durch die gewählte Trainingsmenge.

Beim einfachen Gradientenabstieg gilt  $\theta_{\text{neu}} = \theta_{\text{alt}} - \eta \cdot \text{Gradient}$ . Adam verwendet zusätzlich laufende Schätzungen der ersten und zweiten Gradientenmomente. Für den Einstieg verwenden wir Adam im Gesamtprogramm; den einzelnen skalaren Update rechnen wir zunächst mit SGD nach.

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()
model.train()
for x, y in train_loader:
    optimizer.zero_grad()
    logits = model(x)
    loss = loss_fn(logits, y)
    loss.backward()
    optimizer.step()
```

Ein Epochenloss muss nach der Anzahl der Beispiele gewichtet werden:  $\text{sum}(\text{batch\_loss} \cdot \text{batch\_size}) / \text{total\_examples}$ . Ein ungewichteter Mittelwert der Batchmittelwerte behandelt einen kleineren letzten Batch zu stark. Der Loss muss nicht bei jedem einzelnen Batch sinken.

### Übung 6

1. (15 min) Aktualisieren Sie das skalare Beispiel aus Woche 5 einmal mit  $\eta = 0,1$ . Berechnen Sie den neuen Loss.
2. (30 min) Ergänzen Sie eine Trainingsschleife für das lineare Modell, zunächst auf 2000 Beispielen.
3. (25 min) Protokollieren Sie Trainings- und Validierungsloss pro Epoche. Prüfen Sie an kopierten Parametern, ob tatsächlich ein Update erfolgt.
4. (20 min) Vergleichen Sie Lernraten 0,0001 und 0,01 bei ansonsten gleichen Bedingungen und getrennten Ausgabeverzeichnissen.

Kontrolle: w wird 2,2 und der neue skalare Loss 0,36. Wenigstens ein Parameter ändert sich nach einem gültigen Update. Eine bestimmte MNIST-Accuracy ist hier kein Funktionstest. Zusatz: Tauschen Sie Adam gegen SGD und beschreiben Sie das beobachtete Verhalten.

## 7. Evaluation und Konfusionsmatrix

### Konzept

Zur Evaluation verwenden wir `model.eval()` und `torch.inference_mode()`. Ersteres stellt das Verhalten betroffener Module wie Dropout um. Letzteres verhindert den Aufbau des Gradienten-Rechenwegs. Beide erfüllen unterschiedliche Aufgaben. Die Validierung enthält keinen Optimierschritt.

Accuracy ist der Anteil richtiger Vorhersagen. In einer Konfusionsmatrix  $C$  zählen Zeilen die tatsächliche Klasse und Spalten die vorhergesagte Klasse. Die Diagonale enthält Treffer. Für Klasse  $k$  gilt  $\text{Recall} = C_{kk} / \text{Zeilensumme}_k$  und  $\text{Precision} = C_{kk} / \text{Spaltensumme}_k$ . Bei Nenner null muss die Auswertung eine ausdrücklich dokumentierte Regel nutzen, beispielsweise „nicht definiert“. Ein einzelner Gesamtwert verdeckt systematische Probleme.

### Übung 7

1. (20 min) Berechnen Sie eine Matrix für wahre Klassen  $[0, 0, 1, 1, 2, 2]$  und Vorhersagen  $[0, 1, 1, 1, 0, 2]$ .
2. (30 min) Implementieren Sie eine Evaluationsfunktion ohne Gradienten. Berechnen Sie Loss, Accuracy und Matrix.
3. (20 min) Prüfen Sie `matrix.sum() == Anzahl_Beispiele` und `matrix.diag().sum()/matrix.sum() == accuracy`.
4. (20 min) Bestimmen Sie die häufigsten Verwechslungen auf der Validierung und betrachten Sie fünf zugehörige Bilder.

Kontrolle: Die kleine Matrix hat Zeilen  $[1, 1, 0]$ ,  $[0, 2, 0]$ ,  $[1, 0, 1]$ . Accuracy ist  $4/6$ ; Recall für Klasse 1 ist 1, Precision  $2/3$ . Zusatz: Diskutieren Sie eine ungleiche Klassenverteilung mit 95 % Nullen und einer konstanten Null-Vorhersage.

## 8. Mehrschichtiges Netz und Nichtlinearität

### Konzept

Unser MLP berechnet zunächst  $h = \text{ReLU}(W_1 \cdot x + b_1)$ , dann  $z = W_2 \cdot h + b_2$ . ReLU setzt negative Eingaben auf null und lässt positive unverändert. Ohne Nichtlinearität ließe sich die Verkettung zweier linearer Schichten wieder als eine einzige affine Abbildung ausdrücken. Erst die Aktivierung erweitert die darstellbaren Funktionen.

Die verborgene Schicht mit 128 Einheiten kombiniert Pixel zu erlernten Merkmalen. Diese müssen nicht direkt als menschlich benannte Formen interpretierbar sein. Mehr Parameter erhöhen mögliche Modellkapazität, aber nicht automatisch die Leistung auf neuen Daten. Ein fairer Vergleich verwendet denselben Datensplit und dieselbe Evaluationsmethode.

```
model = nn.Sequential(nn.Flatten(), nn.Linear(784,128),  
                     nn.ReLU(), nn.Linear(128,10))
```

### Übung 8

1. (20 min) Zeichnen Sie den Datenfluss mit Formen  $(N, 784)$ ,  $(N, 128)$  und  $(N, 10)$ .
2. (25 min) Implementieren Sie das MLP und prüfen Sie Parameterzahl sowie Ausgabeform.
3. (25 min) Trainieren Sie lineares Modell und MLP mit gleichem Trainingsbudget. Dokumentieren Sie Validierungsloss und Accuracy.
4. (20 min) Entfernen Sie ReLU und erklären Sie die strukturelle Folge unabhängig vom gemessenen Ergebnis.

Kontrolle: Das MLP hat  $(784 \times 128 + 128) + (128 \times 10 + 10) = 101770$  Parameter. Zusatz: Vergleichen Sie Breiten 32 und 256. Variieren Sie pro Vergleich zunächst nur eine Eigenschaft.

## 9. Generalisierung und kontrollierte Überanpassung

### Konzept

Überanpassung bedeutet, dass das Modell Besonderheiten des Trainingsbestands lernt, die sich nicht ausreichend auf neue Daten übertragen. Ein fallender Trainingsloss bei gleichzeitig steigendem Validierungsloss ist ein typisches Indiz. Unteranpassung liegt nahe, wenn das Modell bereits auf Trainingsdaten die relevante Struktur schlecht erfasst; auch Optimierungsfehler können ähnliche Symptome verursachen.

Regularisierung kann durch Gewichtsstrafe, Dropout, Datenaugmentation oder die Auswahl eines frühen Checkpoints erfolgen. Gewichtung, Anzahl der Trainingsbeispiele und Modellkapazität wirken zusammen. Wir wählen den besten Checkpoint anhand des niedrigsten Validierungsloss. Das Gesamtprogramm trainiert die gewünschte Epochenzahl vollständig; es implementiert keine vorzeitige Beendigung. Early Stopping mit einer Geduldszahl ist eine Erweiterung.

Bei Dropout werden im Training zufällige Aktivierungen unterdrückt. Bei der Auswertung ist Dropout deaktiviert. Augmentationen verändern Trainingsbilder kontrolliert; die Validierung bleibt unverändert. Spiegelungen von Ziffern sind gewöhnlich keine geeignete Annahme über unveränderte Klassen.

### Übung 9

1. (20 min) Trainieren Sie ein MLP auf einem kleinen Trainingsausschnitt und protokollieren Sie beide Loss-Verläufe über mehrere Epochen.
2. (25 min) Zeichnen Sie die Kurven und markieren Sie die nach Validierungsloss beste Epoche.
3. (25 min) Ergänzen Sie wahlweise `weight_decay=0.0001` beim Optimierer oder `nn.Dropout(0.2)` nach ReLU. Halten Sie alle anderen Bedingungen fest.
4. (20 min) Erklären Sie, weshalb zufällige Testbilder oder Test-Accuracy nicht zur Entscheidung zwischen Varianten genutzt werden dürfen.

Kontrolle: Es wird ein Kurvenvergleich abgegeben, auch wenn Überanpassung im konkreten Lauf nicht deutlich auftritt. Keine Kurve erfinden. Zusatz: Implementieren Sie Early Stopping mit `patience = 3` und laden Sie danach den besten statt des letzten Checkpoints.

# 10. Faltung: räumliche Struktur ausnutzen

## Konzept

Ein voll verbundenes Netz betrachtet zunächst alle Pixel als Merkmalsliste. Eine Faltung verarbeitet lokale Nachbarschaften und verwendet dieselben Filtergewichte an vielen Positionen. Dadurch wird ein Musterdetektor räumlich wiederverwendet. Eine kleine Faltung mit 1 Eingangskanal und 8 Ausgangskanälen lernt acht Merkmalskarten. Max-Pooling wählt in einer Region den größten Wert und reduziert die Auflösung.

Für eine räumliche Achse gilt bei  $\text{dilation} = 1$ :  $\text{Ausgabe} = \text{floor}((\text{Eingabe} + 2 \cdot \text{Padding} - \text{Kernelgröße}) / \text{Stride}) + 1$ . Mit Kernel 3, Padding 1 und Stride 1 bleibt 28 erhalten. Ein  $2 \times 2$ -Pooling mit Stride 2 ergibt 14. Unser Netz bildet  $(N, 1, 28, 28)$  auf  $(N, 8, 28, 28)$ , dann  $(N, 8, 14, 14)$ , dann  $(N, 1568)$  und schließlich  $(N, 10)$  ab.

```
model = nn.Sequential(
    nn.Conv2d(1, 8, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
    nn.Flatten(), nn.Linear(8*14*14, 10))
```

## Übung 10

1. (20 min) Rechnen Sie einen  $2 \times 2$ -Filter auf einer selbst gewählten  $3 \times 3$ -Matrix mit Stride 1 ohne Padding durch.
2. (25 min) Implementieren Sie das CNN und geben Sie nach jeder Schicht die Form aus.
3. (25 min) Vergleichen Sie CNN und MLP auf identischem Split und dokumentieren Sie zusätzlich Parameterzahl und Laufzeit.
4. (20 min) Entfernen Sie Padding und korrigieren Sie die Eingangsgröße der letzten Schicht begründet.

Kontrolle: Referenz-CNN hat  $8 \times (1 \times 3 \times 3 + 1) + 10 \times (1568 + 1) = 15770$  Parameter. Ohne Padding sind räumliche Größen 26 und 13; Flatten liefert 1352 Merkmale. Zusatz: Faltung ist nicht generell vollständig translationsinvariant; diskutieren Sie Ränder und Pooling.

# 11. CPU und CUDA sinnvoll vergleichen

## Konzept

CUDA ermöglicht die Ausführung unterstützter Tensoroperationen auf einer NVIDIA-GPU. Dafür müssen Modellparameter und Eingaben auf dem passenden Gerät liegen. `model.to(device)` verschiebt das Modell, `x.to(device)` liefert einen Tensor auf diesem Gerät; die Rückgabe muss verwendet werden. Eine Mischung von CPU- und GPU-Tensoren führt häufig zu einem Gerätefehler.

Eine GPU ist nicht bei jeder kleinen Aufgabe schneller. Transferkosten, Startkosten und kleine Batches können den Vorteil überdecken. CUDA-Operationen werden häufig asynchron angestoßen. Eine Zeitmessung ohne Synchronisation kann nur das Einstellen der Arbeit statt deren Abschluss messen. Warm-up, mehrere Wiederholungen und identische Aufgaben sind für einen fairen Vergleich nötig.

```
import time
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
x = torch.rand(64,1,28,28,device=device)
model.eval()
with torch.inference_mode():
    for _ in range(10): model(x)
    if device.type == 'cuda': torch.cuda.synchronize()
    start = time.perf_counter()
    for _ in range(100): model(x)
    if device.type == 'cuda': torch.cuda.synchronize()
    elapsed = time.perf_counter()-start
print('Sekunden pro Batch:', elapsed/100)
```

## Übung 11

1. (15 min) Dokumentieren Sie Geräteverfügbarkeit sowie Position von Modell und Eingaben.
2. (30 min) Messen Sie Inferenz bei Batchgrößen 1 und 64, getrennt von Datendownload und Training.
3. (25 min) Vergleichen Sie CPU und CUDA auf demselben Rechner, falls verfügbar. Andernfalls nutzen Sie die CPU-Batchmessung und bereitgestellte Messdaten der Lehrperson.
4. (20 min) Erklären Sie, welche Messung Transfers enthält, welche nur Rechenzeit untersucht und weshalb dies verschiedene Fragestellungen sind.

Kontrolle: Bericht enthält Hardware, Version, Warm-up, Wiederholungen und Messumfang. Keine universelle Aussage aus einem einzigen Lauf. Zusatz: Messen Sie separat Host-zu-Gerät-Transfer und vollständige End-to-End-Inferenz.

## 12. Modell speichern und nach Neustart benutzen

### Konzept

Ein `state_dict` enthält die gelernten Parameter eines Modells. Die zugehörige Architektur und Vorverarbeitung müssen bekannt bleiben. Im Kurscheckpoint stehen deshalb Architekturname und Gewichte. Für echte Fortsetzung des Trainings wären zusätzlich Optimiererzustand, Epochenstand und weitere Metadaten erforderlich; der Kurscheckpoint ist für Inferenz und Auswahl des besten Modells gedacht.

Wir laden eigene vertrauenswürdige Checkpoints mit `weights_only=True`, instanziiieren die passende Architektur und rufen `load_state_dict` auf. `map_location` erlaubt CPU-Inferenz eines ursprünglich auf GPU gespeicherten Modells. Danach setzen wir den Auswertungsmodus. Ein einzelnes Bild benötigt eine ergänzte Batchachse.

```
state = torch.load('run/best.pt', map_location='cpu', weights_only=True)
model = model_for(state['model'])
model.load_state_dict(state['state_dict'])
model.eval()
with torch.inference_mode():
    logits = model(image.unsqueeze(0))
    predicted = logits.argmax(1).item()
```

### Übung 12

1. (20 min) Speichern Sie den besten Validierungcheckpoint samt Architekturkennung.
2. (25 min) Beenden Sie Python, laden Sie in einem neuen Prozess und berechnen Sie dieselben Vorhersagen.
3. (25 min) Vergleichen Sie Logits vor und nach dem Laden mit `torch.allclose`; verwenden Sie dasselbe Gerät und dieselbe Eingabe.
4. (20 min) Erstellen Sie eine kurze Modellbeschreibung: Datenformat, Vorverarbeitung, Trainingsdaten, Auswahlkriterium und bekannte Grenzen.

Kontrolle: Gleichheit wird für unveränderte gespeicherte Parameter und feste Eingaben geprüft, nicht für zwei unabhängig neu trainierte Modelle. Zusatz: Bauen Sie einen trainierbaren Fortsetzungcheckpoint mit Optimiererzustand.

# 13. Eigene Bilder und Grenzen des Systems

## Konzept

MNIST-Ziffern besitzen einen weitgehend einheitlichen Stil: Graustufen, dunkler Hintergrund, heller Strich und passend platzierte Ziffern. Eine Smartphoneaufnahme von schwarzer Schrift auf weißem Papier ist eine andere Datenverteilung. Eine bloße Größenänderung kann deshalb schlechte Ergebnisse liefern. Vorverarbeitung muss zum Trainingsformat passen und nachvollziehbar beschrieben werden.

Erstellen Sie zunächst Bilder mit heller Ziffer auf schwarzem Hintergrund, großzügigem Rand und ungefähr zentrierter Form. Die Referenz liest ein Bild als Graustufen und skaliert es auf  $28 \times 28$ ; sie führt keine automatische Segmentierung, Zentrierung oder Invertierung aus. Später können Sie solche Schritte ergänzen. Ein leerer Hintergrund oder ein Buchstabe erhält beim Zehnklassenmodell trotzdem eine Ziffer als  $\text{argmax}$ . Das Modell hat keine ausdrücklich trainierte Klasse „keine Ziffer“.

Ein Softmax-Maximum von 0,99 ist keine Garantie für 99 % korrekte Vorhersagen bei beliebigen Eingaben. Eine Ablehnungsschwelle braucht eigene Validierung auf geeigneten Beispielen. Für personenbezogene Anwendungsdaten sind Datensparsamkeit, Einwilligung beziehungsweise Rechtsgrundlage und das Vermeiden unnötiger Speicherung zu berücksichtigen. Bei unserem Übungsprojekt werden keine Namen oder Unterschriften erfasst.

## Übung 13

1. (20 min) Erzeugen Sie zehn eigene Ziffernbilder und legen Sie deren richtige Klassen vor der Auswertung fest.
2. (25 min) Testen Sie die gespeicherte Lösung. Dokumentieren Sie Treffer und Fehler; behandeln Sie die zehn Bilder als kleine explorative Stichprobe.
3. (25 min) Variieren Sie Position, Hintergrund und Strichdicke kontrolliert. Prüfen Sie zusätzlich Leerbild und einen Buchstaben.
4. (20 min) Schlagen Sie eine Vorverarbeitungsverbesserung vor und entwickeln Sie eine getrennte Prüfmenge dafür.

Kontrolle: Fehler werden mit Eingangsbildern und Vorverarbeitung dokumentiert. Die zusätzliche Stichprobe liefert keinen Nachweis allgemeiner Alltagstauglichkeit. Zusatz: Entwickeln Sie ein Ablehnungsverfahren und messen Sie sowohl Fehlablehnungen als auch fälschlich akzeptierte Nichtziffern.

# 14. Abschluss: ein Ergebnis wissenschaftlich vertreten

## Konzept

Ein belastbares Projekt beantwortet eine begrenzte Frage mit einem nachvollziehbaren Versuch. Unser Abschluss fragt: Welche der drei Architekturen liefert unter einem festgelegten Budget die beste Validierungsleistung, und welche Grenzen zeigt die gewählte Lösung? Die Entscheidung muss vor dem finalen Test feststehen. Ein hoher Testwert kompensiert keine Datenleckage oder unklare Versuchsbedingungen.

Der Bericht unterscheidet Beobachtung und Erklärung. „Das CNN erzielte in diesem Lauf eine höhere Accuracy“ ist eine Beobachtung. „Das liegt sicher an besserer räumlicher Merkmalsnutzung“ ist eine Erklärung, die weitere kontrollierte Versuche benötigt. Wiederholen Sie zentrale Versuche möglichst mit drei Trainingsseeds bei identischem Datensplit und berichten Sie Mittelwert sowie Streuung. Reproduzierbarkeit bedeutet auch, dass ein anderer Studierender den Versuch anhand der Dokumentation ausführen kann.

## Übung 14 / Präsentation

1. (15 min) Frieren Sie Architektur, Hyperparameter und Vorverarbeitung anhand der Validierung ein.
2. (25 min) Führen Sie den finalen Test aus und protokollieren Sie Matrix, Loss und Accuracy.
3. (30 min) Präsentieren Sie in kleinen Gruppen Fragestellung, Vergleich, drei Fehlerbeispiele und Grenzen.
4. (20 min) Beantworten Sie Rückfragen zu Batchform, Kreuzentropie, Validierung und Geräteausführung.

Abgabe: Code, konkrete Paketliste, Versuchskonfigurationen, Messwerte und Bericht von etwa fünf bis acht Seiten. Vor dem finalen Test wurde das Auswahlkriterium dokumentiert. Der Testbestand wird nicht zur nachträglichen Optimierung verwendet. Zusatz: Lassen Sie eine andere Gruppe den ausgewählten Versuch reproduzieren.

## Referenzprogramm: Bedienung und Zuordnung

```
python kurs.py --smoke
python kurs.py --model linear --epochs 3 --out run_linear
python kurs.py --model mlp --epochs 3 --out run_mlp
python kurs.py --model cnn --epochs 3 --out run_cnn
```

Die drei Aufrufe sind Beispiele, keine Festlegung eines optimalen Trainingsbudgets. Für erste Übungen verwenden Sie `--limit 2000 --epochs 1`. Jeder Lauf erzeugt `best.pt` und `metrics.json`. Der feste Splitseed 2026 bleibt für alle Versuche gleich; `--seed` ändert Initialisierung und Trainingsreihenfolge. Vergleichen Sie Versuche mit passenden, ausdrücklich festgelegten Budgets. Für die Abschlussauswahl kann das gleiche Epochenbudget oder eine gemessene Zeitgrenze untersucht werden, aber beides beantwortet unterschiedliche Fragen.

Nach der Modellwahl kann ein vorab festgelegter endgültiger Trainingslauf einschließlich Test erfolgen, beispielsweise:

```
python kurs.py --model cnn --epochs 3 --seed 42 --out final --test
python kurs.py --out final --image eigene_7.png
```

`--test` wertet am Ende den in diesem Lauf nach Validierungsloss besten Checkpoint aus. Verwenden Sie die Option nicht während der Modellsuche. `--image` lädt den bereits gespeicherten Checkpoint aus `--out`; es startet kein Training. Für CUDA ergänzen Sie `--device cuda` oder `--device auto`. Der

Standard ist CPU. Dateien in einem schon benutzten Ausgabeordner können überschrieben werden; wählen Sie pro Versuch einen eigenen Ordner.

| Bestandteil von kurs.py                                    | Einführung im Semester |
|------------------------------------------------------------|------------------------|
| Tensorformen in <code>smoke()</code>                       | Woche 2                |
| MNIST, <code>random_split</code> , <code>DataLoader</code> | Woche 3                |
| <code>model_for('linear')</code>                           | Woche 4                |
| <code>cross_entropy</code> , <code>backward</code>         | Woche 5                |
| Adam, <code>zero_grad</code> , <code>step</code>           | Woche 6                |
| <code>evaluate()</code> , Konfusionsmatrix                 | Woche 7                |
| <code>model_for('mlp')</code>                              | Woche 8                |
| Auswahl von <code>best.pt</code>                           | Woche 9                |
| <code>model_for('cnn')</code>                              | Woche 10               |
| Geräteauswahl und <code>.to(device)</code>                 | Woche 11               |
| <code>torch.save/load</code> , <code>--image</code>        | Woche 12/13            |
| <code>--test</code> und Versuchsdokumentation              | Woche 14               |

## Lösungshinweise und häufige Fehler

Woche 1-3: Accuracy zählt Gleichheiten und teilt durch die Zahl der Beispiele. Bei leeren Listen sollte die selbst entwickelte Funktion etwa `ValueError` auslösen. Die Baseline der Liste aus Woche 1 sagt 0 und trifft 3/5. Für `x[0]` fehlt nur die Batchachse, nicht die Kanalachse. Splitmengen werden anhand ihrer ursprünglichen Indizes überprüft. Eine neue zufällige Aufteilung pro Modell würde Vergleiche erschweren.

Woche 4-6: `nn.Linear` speichert Gewichtsmatrizen mit Ausgangsmerkmalen als erster Achse. Für den Handversuch setzen Sie `layer.weight.copy_(W)` unter `torch.no_grad()` und den Bias auf null. Kreuzentropie erhält Klassenindizes statt One-Hot-Vektoren im hier verwendeten Modus. Nach `backward()` stehen Gradienten in `parameter.grad`, aber erst `step()` verändert Parameter. Fehlendes `zero_grad()` führt zu unerwünschter Akkumulation.

Woche 7-9: Zeilen und Spalten der Matrix dürfen nicht vertauscht werden. `eval()` allein schaltet die Gradientenberechnung nicht ab. Die Auswahl des besten Checkpoints erfolgt über Validierungsloss und wird vor einem Testlauf festgelegt. Für einen Regularisierungsvergleich werden neue Modelle mit denselben Startbedingungen trainiert; das bereits trainierte Modell einfach weiterlaufen zu lassen wäre kein fairer Neustartvergleich.

Woche 10-12: In der CNN-Parameterzahl gehören Biaswerte dazu. Die Formänderung durch Pooling muss vor der letzten linearen Schicht berücksichtigt werden. CUDA-Zeitmessung benötigt Synchronisation; Datentransfer und reine Berechnung werden getrennt benannt. Laden Sie dieselbe Architektur wie beim Speichern. Vergessen Sie nicht `eval()` und die Batchachse.

Woche 13-14: Ein invertiertes Bild kann trotz korrekter Form außerhalb der Trainingsverteilung liegen. Für neue Vorverarbeitung brauchen Sie einen eigenen Validierungsversuch. Keine Wahrscheinlichkeit als Gewissheit ausgeben. Reportieren Sie auch ungünstige oder unerwartete Resultate und unterscheiden Sie Formtests von Qualitätsmessungen.

## Kurzer mathematischer Anhang

Skalarprodukt: Für  $x$  und  $w$  gleicher Länge ist  $w \cdot x = \sum_j w_j x_j$ . Jede Klasse im linearen Modell besitzt einen eigenen Gewichtsvektor.

Softmax:  $p_k = \exp(z_k) / \sum_j \exp(z_j)$ . Für eine numerisch stabilere Berechnung kann vor der Exponentialfunktion überall derselbe größte Logit abgezogen werden. Nutzen Sie in PyTorch die vorhandene Implementierung.

Kreuzentropie: Für eine Zielklasse  $y$  gilt  $L = -\log(p_y)$ . Im Kursfall ergibt sich für den Logit  $z_k$  die Ableitung  $p_k - 1$ , wenn  $k = y$ , sonst  $p_k$ . Für den Batchmittelwert wird zusätzlich durch die Batchgröße geteilt.

Kettenregel: Hängt  $L$  von  $h$  und  $h$  von  $w$  ab, gilt  $dL/dw = (dL/dh) \cdot (dh/dw)$ . Backpropagation wendet diese Regel rückwärts durch den Rechenweg an. Bei ReLU ist die Ableitung außerhalb null für positive Eingaben eins und für negative null; an null verwendet PyTorch eine festgelegte Subgradientenwahl.

Precision/Recall: Precision fragt, welcher Anteil der als  $k$  ausgegebenen Beispiele tatsächlich  $k$  ist. Recall fragt, welcher Anteil der tatsächlichen  $k$ -Beispiele erkannt wurde. Für die Beurteilung unter Klassenungleichgewicht können Klassenmittelwerte zusätzlich zur Accuracy sinnvoll sein.

## Begriffe zum Nachschlagen

Batch: gemeinsam verarbeitete Beispiele. Epoche: Durchlauf durch Trainingsdaten. Parameter: gelernte Gewichte/Biaswerte. Hyperparameter: vorab gewählte Einstellungen wie Lernrate. Logit: unnormalisierter Klassenscore. Loss: zu minimierende Zielfunktion. Gradient: lokale Ableitung nach Parametern. Inferenz: Anwendung des trainierten Modells. Validierung: Daten für Modellentscheidungen. Test: zurückgehaltene Daten für die abschließende Bewertung. Datenleckage: Informationen aus späteren Bewertungsdaten beeinflussen unerlaubt Training oder Auswahl. Generalisierung: Verhalten auf neuen passenden Beispielen.

## Offizielle Quellen und Vertiefung

Die Unterlagen sind eigenständig formuliert. Für Installation und API-Verhalten dienen die offiziellen Dokumentationen als Referenz; prüfen Sie die zu Ihrer eingesetzten Version gehörende Dokumentation.

1. Installation: <https://pytorch.org/get-started/locally/>
2. Grundlagen und vollständiger Arbeitsablauf:  
<https://docs.pytorch.org/tutorials/beginner/basics/intro.html>
3. MNIST-Schnittstelle:  
<https://docs.pytorch.org/vision/stable/generated/torchvision.datasets.MNIST.html>
4. Kreuzentropie: <https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>
5. Reproduzierbarkeit: <https://docs.pytorch.org/docs/stable/notes/randomness.html>
6. Modellgewichte speichern:  
[https://docs.pytorch.org/tutorials/beginner/basics/saveloadrun\\_tutorial.html](https://docs.pytorch.org/tutorials/beginner/basics/saveloadrun_tutorial.html)

Hinweis für Lehrende: Alle Zahlen zu Parameteranzahl, Formen und kleinen Handbeispielen sind analytische Sollwerte. Trainingsqualität und Laufzeiten sind experimentell zu ermitteln. Planen Sie vor Semesterbeginn einen CPU-Probelauf, Datenbereitstellung und die passende Paketinstallation; passen Sie Zeitbudgets bei langsamen Rechnern durch kleinere Trainingsausschnitte an.