Künstliche Intelligenz selber programmieren mit PyTorch

Peter Bouda

Veröffentlicht von AI Workshops
www.aiworkshops.eu

Version 1.0

Inhaltsverzeichnis

Einleitung

Dieses Buch führt Sie in die Entwicklung von Deep-Learning-Modellen mit Python und PyTorch ein. Die Zielgruppe ist bewusst weit gefasst: Interessierte Laien mit Programmierkenntnissen sollen sich ebenso angesprochen fühlen wie erfahrene Python-Entwickler, die in die Welt der künstlichen Intelligenz einsteigen möchten. PyTorch bietet einen entscheidenden Vorteil: Die enge Verzahnung mit Python ermöglicht einen intuitiven Einstieg in neuronale Netze, während die Bibliothek gleichzeitig die Werkzeuge für professionelle Anwendungen bereitstellt. Aber auch die reine Freude am Experimentieren soll in diesem Buch nicht zu kurz kommen, schließlich ist es faszinierend zu verstehen, wie moderne KI-Systeme wie ChatGPT oder Bildgeneratoren tatsächlich funktionieren.

Für die Lektüre sollten Sie zumindest die Grundlagen der Python-Programmierung beherrschen. Wenn Sie in der Lage sind, eine Klasse in Python zu implementieren und mit Listen und Dictionaries zu arbeiten, haben Sie diese Hürde schon gemeistert und können sofort mit den Beispielen in diesem Buch starten. Mathematische Vorkenntnisse sind hilfreich, aber nicht zwingend erforderlich, die notwendigen Konzepte werden im Kontext erklärt. Beachten Sie aber, dass Sie hier keine PyTorch-Referenz vor sich haben. Dieses Buch kann und möchte Ihnen nicht das PyTorch-Framework mit all seinen Klassen und Modulen vorstellen. Vielmehr werden Sie Schritt für Schritt in die Entwicklung neuronaler Netze sowie in die grundlegenden Konzepte des Deep Learning eingeführt. Für die weitere Arbeit empfiehlt es sich dann, auf die offizielle PyTorch-Dokumentation zurückzugreifen.

Ein Schwerpunkt dieses Buches liegt auf dem praktischen Verständnis: Wie funktionieren neuronale Netze wirklich? Was passiert beim Training, und warum? Anstatt nur fertige Bausteine zusammenzusetzen, werden Sie viele Konzepte zunächst von Grund auf implementieren, bevor Sie die entsprechenden PyTorch-Funktionen kennenlernen. Dieses Verständnis ist unerlässlich, wenn Sie später eigene Modelle entwickeln oder bestehende Architekturen an Ihre Bedürfnisse anpassen möchten.

Das Buch ist in fünf Kapitel gegliedert, die aufeinander aufbauen. Im ersten Kapitel lernen Sie PyTorch und Tensoren kennen – das Fundament für alles Weitere. Das zweite Kapitel führt Sie durch die Grundlagen neuronaler Netze, vom einfachen Perzeptron bis zu mehrschichtigen Netzwerken mit dem `torch.nn`-Modul. Im dritten Kapitel treffen Sie auf spezialisierte Architekturen: Convolutional Neural Networks für die Bildverarbeitung und Recurrent Neural Networks für sequenzielle Daten wie Text. Das vierte Kapitel vertieft die Sequenzverarbeitung mit LSTM-Zellen, Encoder-Decoder-Modellen und dem revolutionären Attention-Mechanismus. Im fünften Kapitel schließlich implementieren Sie die Transformer-Architektur, das Herzstück moderner Large Language Models, und erhalten einen Ausblick auf BERT, GPT und die Technologien, die heute die KI-Landschaft prägen.

Die Kapitel sollten Sie nacheinander lesen, da die Code-Beispiele aufeinander aufbauen. Die Beispiele der ersten Kapitel sind bewusst einfach gehalten; die Möglichkeiten und Einsatzgebiete der einzelnen Technologien werden sukzessive im Verlauf des Buches erweitert. Am Ende werden Sie nicht nur verstehen, wie moderne KI-Systeme funktionieren, sondern auch die Werkzeuge besitzen, um eigene Modelle zu entwickeln und mit den neuesten Architekturen zu experimentieren.

Noch ein Hinweis: Beim Schreiben dieses Buches hatte ich Unterstützung von KI. Ich habe das Konzept und den Aufbau für einen PyTorch-Kurs entwickelt, und daraus dann einen ersten Entwurf für das Buch geschrieben. Die KI habe ich dann als Reviewer genutzt, um Vorschläge zu Verbesserungen bei Formulierungen und dem inhaltlichen Aufbau zu bekommen. Dabei sind mir auch wieder die noch vorhandenen Einschränkungen bei KI-Nutzung bewusst geworden: einige der Vorschläge waren dann doch zu unverständlich oder zu verkürzt für eine Einführung. Zusammen mit meinem “Sparring Partner” habe ich aber letztendlich ein hoffentlich hilfreiches Buch zur Einführung in moderne KI-Implementierungen erstellt.

Der Online-Kurs mit dem Inhalt dieses Buches ist hier verfügbar (auf Englisch); für alle, die gerne noch mit selbst gewähltem Tempo und praktischen Übungen vertieft in die Materie eintauchen wollen:

https://tech-academy.pt/curso/deep-learning-with-pytorch/

Zuletzt bleibt mir noch, Ihnen viel Freude bei der Lektüre dieses Buches und beim Experimentieren mit künstlicher Intelligenz zu wünschen!

Bei Fragen oder Anregungen schicken Sie mir einfach eine Mail an:

pbouda@outlook.com

Code zum Buch

Alle Code-Beispiele in diesem Buch sind in einem Github-Repository verfügbar:

https://github.com/aiworkshops-eu/pytorch-course

Spätestens ab Kapitel 4 werden die Code-Beispiele immer komplexer, so dass ich nicht mehr den gesamten Code für ein komplettes Beispiel am Stück präsentieren kann. Schauen Sie deswegen auch unbedingt den entsprechenden Code im Repository an, das kann beim Verständnis helfen und Sie können das gesamte Modelltraining im Überblick behalten. Ich empfehle dann auch, den Code einfach mal Laufen zu lassen um die Ausgabe zu verfolgen. Natürlich sind Sie auch herzlich eingeladen, den Code zu editieren und gegebenenfalls Fragen und Anregungen zu schicken. Nutzen Sie dazu einfach die Issues auf Github!

Über AI Workshops

Mit AI Workshops biete ich Kurse und Beratung für KI-Technologien an. Die Workshops sind online oder offline, mit angepassten Inhalten für Ihre Bedürfnisse und passend zu Ihrem Zeitplan.

Neben den Kursen biete ich auch personalisierte Services wie begleitetes Vibe Coding oder Unterstützung bei On-Premise KI-Deployments. Die Idee dahinter: Ich bringe Sie in 1-4 Stunden auf den aktuellen Stand der Technik und begleite Sie dann bei der Implementierung und dem Ausrollen. Mit individuellen Sessions für Einzelpersonen oder kleine Teams. Gerne auch mit Inhalten über PyTorch :)

Mein Angebot finden Sie hier: https://www.aiworkshops.eu

Kapitel 1: PyTorch-Grundlagen und Tensoren

In diesem ersten Kapitel werden Sie in die Welt von PyTorch eingeführt. PyTorch ist ein Open-Source-Framework für maschinelles Lernen, das sich durch seine intuitive Handhabung und Flexibilität auszeichnet. Sie werden lernen, was Tensoren sind, wie Sie diese erstellen und manipulieren, und wie PyTorch automatisch Gradienten berechnet: die Grundlage für das Training neuronaler Netze.

1.1 Einführung in PyTorch

Was ist PyTorch?

PyTorch ist ein Open-Source-Framework für maschinelles Lernen, das ursprünglich von Meta (ehemals Facebook) entwickelt wurde. Es hat sich in den letzten Jahren zu einem der beliebtesten Werkzeuge in der KI-Forschung und -Entwicklung etabliert. Was PyTorch besonders auszeichnet, ist seine Python-nahe, intuitive API und die Möglichkeit, Berechnungen dynamisch aufzubauen.

Im Gegensatz zu anderen Frameworks verwendet PyTorch sogenannte *dynamische Berechnungsgraphen* (Englisch: “dynamic computational graph”). Das bedeutet, dass der Graph der mathematischen Operationen während der Ausführung erstellt wird, also “define-by-run”. Dies macht das Debuggen erheblich einfacher, da Sie Standard-Python-Werkzeuge wie `print()` oder den Debugger Ihrer Entwicklungsumgebung verwenden können, um zu verstehen, was in Ihrem Netzwerk passiert.

PyTorch vs. andere Frameworks

Falls Sie bereits von anderen Frameworks wie TensorFlow, JAX oder Keras gehört haben, fragen Sie sich vielleicht, warum Sie sich für PyTorch entscheiden sollten. Hier ein kurzer Vergleich:

- **PyTorch** bietet dynamische Graphen, einfacheres Debugging und wird besonders in der Forschung geschätzt. Die meisten aktuellen wissenschaftlichen Veröffentlichungen im Bereich Deep Learning verwenden PyTorch.
- **TensorFlow** war historisch auf statische Graphen ausgelegt und ist stark auf Produktionsumgebungen fokussiert.
- **JAX** verfolgt einen funktionalen Programmieransatz mit starker XLA-Kompilierung.
- **Keras** bietet eine High-Level-API und ist mittlerweile in TensorFlow integriert.

Für den Einstieg in Deep Learning ist PyTorch eine hervorragende Wahl, da die Konzepte direkt in Python umgesetzt werden und Sie nicht erst eine neue "Sprache innerhalb der Sprache" lernen müssen.

Installation und Setup

Für die Installation von PyTorch empfehle ich die Verwendung von uv, einem modernen Python-Paketmanager, der die Verwaltung von Abhängigkeiten erheblich vereinfacht. Falls Sie uv noch nicht installiert haben, können Sie es unter Linux und macOS mit folgendem Befehl installieren:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Unter Windows verwenden Sie PowerShell:

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Erstellen Sie zunächst einen Projektordner und initialisieren Sie dort ein neues Python-Projekt:

```
uv init mein-pytorch-projekt
cd mein-pytorch-projekt
```

Die Installation von PyTorch erfolgt dann mit einem einzigen Befehl:

```
uv add torch torchvision torchaudio
```

Nach der Installation können Sie die erfolgreiche Einrichtung mit einem kleinen Python-Skript überprüfen:

```
import torch

# PyTorch-Version anzeigen
print(f"PyTorch version: {torch.__version__}")

# Prüfen, ob CUDA (GPU-Unterstützung) verfügbar ist
print(f"CUDA available: {torch.cuda.is_available()}")

# Zufallszahlen-Generator initialisieren für reproduzierbare Ergebnisse
torch.manual_seed(42)

print("Environment setup confirmed!")
```

Wenn Sie dieses Skript ausführen und keine Fehlermeldung erscheint, ist Ihre Umgebung korrekt eingerichtet.

Benötige ich eine GPU?

Eine häufig gestellte Frage ist, ob man für das Erlernen von PyTorch eine Grafikkarte benötigt. Die kurze Antwort: Nein, für die Grundlagen und kleinere Projekte reicht ein normaler Prozessor (CPU) vollkommen aus.

Eine GPU beschleunigt das Training neuronaler Netze erheblich, je nach Modell und Datensatz um das 10- bis 100-fache. Dieser Geschwindigkeitsvorteil wird jedoch erst bei größeren Modellen und Datensätzen relevant. Alle Beispiele in diesem Buch laufen problemlos auf einer CPU. Wenn Sie später mit größeren Bilddatensätzen, langen Textsequenzen oder tieferen Netzwerken arbeiten möchten,

können Sie auf Cloud-Dienste wie Google Colab oder Kaggle zurückgreifen, die kostenlose GPU-Ressourcen anbieten.

1.2 Deep Learning in einfachen Worten

Bevor wir uns den Details von PyTorch widmen, sollten wir verstehen, was Deep Learning eigentlich bedeutet und wie es funktioniert.

Lineare und nicht-lineare Transformationen

Im Kern besteht Deep Learning aus einer einfachen Idee: Wir stapeln viele einfache mathematische Operationen übereinander, um komplexe Probleme zu lösen. Das Grundmuster sieht dabei immer gleich aus:

1. **Lineare Transformation**: Eine Matrixmultiplikation, die Sie vielleicht aus der Schule als `y = ax + b` kennen. Im mehrdimensionalen Fall wird daraus eine Matrixmultiplikation.
2. **Nicht-lineare Aktivierung**: Eine Funktion wie ReLU oder Sigmoid, die dafür sorgt, dass das Netzwerk auch nicht-lineare Zusammenhänge lernen kann. Namen wie `ReLU` klingen vielleicht zunächst etwas abschreckend mathematisch, oft sind die mathematischen Funktionen aber relativ einfach. In diesem Fall bedeutet `ReLU`: Wenn der Wert größer null ist, dann gebe ihn weiter, ansonsten gebe null weiter.
3. **Wiederholen**: Diese beiden Schritte werden mehrfach hintereinander ausgeführt.

Warum brauchen wir die nicht-lineare Aktivierung? Stellen Sie sich vor, Sie würden nur lineare Transformationen hintereinander ausführen. Mathematisch gesehen wäre das Ergebnis immer noch eine einzige lineare Transformation – egal wie viele Schichten Sie stapeln. Erst die nicht-lineare Aktivierung ermöglicht es dem Netzwerk, komplexe Muster zu erkennen. Eine der nicht besonders intuitiven Einsichten in Deep Learning ist, dass solche nicht-linearen Funktion das Lernen beliebiger Funktionen ermöglichen.

Layer-Architektur: Der Aufbau neuronaler Netze

Ein neuronales Netz besteht aus mehreren *Schichten* (Layers). Jede Schicht verarbeitet die Eingabe und gibt sie an die nächste Schicht weiter. Bei der Bildverarbeitung lernen die ersten Schichten typischerweise einfache Merkmale wie Kanten und Linien zu erkennen. Die mittleren Schichten kombinieren diese zu komplexeren Formen, und die letzten Schichten erkennen schließlich ganze Objekte.

Was sind Tensoren?

Der Begriff “Tensor” stammt ursprünglich aus der Physik und Mathematik, wo er multidimensionale Datenobjekte beschreibt. In PyTorch können Sie sich Tensoren als eine Verallgemeinerung von Vektoren und Matrizen vorstellen:

- **0D-Tensor (Skalar)**: Eine einzelne Zahl, z.B. `5`
- **1D-Tensor (Vektor)**: Eine Liste von Zahlen, z.B. `[1, 2, 3]`
- **2D-Tensor (Matrix)**: Eine Tabelle von Zahlen, z.B. `[[1, 2], [3, 4]]`
- **3D+ Tensor**: Höherdimensionale Arrays, z.B. ein Stapel von Bildern

PyTorch-Tensoren sind jedoch mehr als nur mehrdimensionale Arrays. Sie können auf der GPU berechnet werden und, was für das Training neuronaler Netze entscheidend ist, sie können Gradienten verfolgen. Diese Gradienten werden wir verwenden, um die Ableitung (bzw. das Ergebnis der Ableitung für den Eingabewert) von Funktionen in den einzelnen Schichten zu bestimmen.

1.3 Tensoren erstellen und verwenden

Nachdem Sie nun verstehen, was Tensoren sind, werden wir praktisch. In diesem Abschnitt lernen Sie, wie Sie Tensoren in PyTorch erstellen und deren Eigenschaften abfragen.

Von Python-Listen zu Tensoren

Der einfachste Weg, einen Tensor zu erstellen, ist die Konvertierung einer Python-Liste:

```
import torch

# Tensor aus einer verschachtelten Liste erstellen
list_data = [[1, 2], [3, 4], [5, 6]]
tensor_from_list = torch.tensor(list_data)
print("Tensor aus Liste:\n", tensor_from_list)
```

Die Ausgabe zeigt einen 2D-Tensor (eine Matrix) mit 3 Zeilen und 2 Spalten:

```
Tensor aus Liste:
 tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

Vektoren und Matrizen aus der Schule

Falls Sie sich noch an Vektoren und Matrizen aus dem Mathematikunterricht erinnern, werden Ihnen PyTorch-Tensoren vertraut vorkommen.

Ein **Vektor** ist eine geordnete Liste von Zahlen. In der Schule haben Sie Vektoren vielleicht für Positionen oder Geschwindigkeiten verwendet. In PyTorch erstellen Sie einen Vektor so:

```
# Vektor erstellen
vector = torch.tensor([3, 4])
print("Vektor:", vector)
```

Operationen wie Addition oder Skalierung (Multiplikation einer Zahl mit einem Vektor) funktionieren wie erwartet:

```
# Vektoraddition
v1 = torch.tensor([1, 2])
v2 = torch.tensor([3, 4])
print("Addition:", v1 + v2)  # Ausgabe: tensor([4, 6])

# Skalierung
print("Skalierung:", 2 * v1)  # Ausgabe: tensor([2, 4])
```

Eine **Matrix** ist eine rechteckige Anordnung von Zahlen in Zeilen und Spalten:

```
# Matrix erstellen
matrix = torch.tensor([[1, 2], [3, 4]])
print("Matrix:\n", matrix)
```

Tensor-Eigenschaften: Shape, dtype, device

Jeder Tensor hat wichtige Eigenschaften, die Sie abfragen können:

```
import torch

# Beispiel-Tensor erstellen (2 Blöcke/Batches mit je 3 Zeilen und 4 Spalten)
sample_tensor = torch.tensor([
    [[1.0, 2.0, 3.0, 4.0],
     [5.0, 6.0, 7.0, 8.0],
     [9.0, 10.0, 11.0, 12.0]],

    [[13.0, 14.0, 15.0, 16.0],
     [17.0, 18.0, 19.0, 20.0],
     [21.0, 22.0, 23.0, 24.0]]
])

print("Shape:", sample_tensor.shape)        # torch.Size([2, 3, 4])
print("Datentyp:", sample_tensor.dtype)     # torch.float32
print("Gerät:", sample_tensor.device)       # cpu
print("Anzahl Dimensionen:", sample_tensor.ndim)  # 3
print("Gesamtzahl Elemente:", sample_tensor.numel())  # 24
```

Die **Shape** gibt an, wie viele Elemente jede Dimension enthält. Ein Tensor mit Shape `(2, 3, 4)` hat 2 Elemente in der ersten Dimension, 3 in der zweiten und 4 in der dritten.

Der **Datentyp** (dtype) definiert, wie die Zahlen im Speicher abgelegt werden. Die häufigsten Typen sind:

- `torch.float32`: 32-Bit-Gleitkommazahlen (Standard für die meisten Operationen)
- `torch.int64`: 64-Bit-Ganzzahlen (Standard für Integer-Tensoren)
- `torch.bool`: Boolesche Werte (True/False)

Das **Gerät** (device) gibt an, ob der Tensor auf der CPU oder GPU liegt.

Tensor-Initialisierung: zeros, ones, randn, arange

PyTorch bietet verschiedene Funktionen, um Tensoren mit bestimmten Werten zu initialisieren. Die Parameter dieser Funktionen definieren den *Shape* (die Form) des Tensors, also die Größe jeder Dimension. Der Aufruf `torch.zeros(3, 4)` erstellt beispielsweise einen Tensor mit 3 Zeilen und 4 Spalten, also mit Shape `(3, 4)`. Bei `torch.randn(2, 3, 4)` entsteht ein 3D-Tensor mit Shape `(2, 3, 4)`, also 2 “Blöcke” mit je 3 Zeilen und 4 Spalten:

```
import torch

# Tensor mit Nullen
```

```
zeros_tensor = torch.zeros(3, 4)
print("Nullen:\n", zeros_tensor)

# Tensor mit Einsen
ones_tensor = torch.ones(2, 3)
print("Einsen:\n", ones_tensor)

# Tensor mit Zufallswerten aus der Normalverteilung
random_tensor = torch.randn(2, 2)
print("Zufallswerte:\n", random_tensor)

# Tensor mit Zahlenfolge
range_tensor = torch.arange(0, 12, 2)  # Start, Ende, Schrittweite
print("Zahlenfolge:", range_tensor)  # tensor([0, 2, 4, 6, 8, 10])

# Tensor mit spezifischem Datentyp
float_tensor = torch.tensor([1, 2, 3], dtype=torch.float32)
print("Float-Tensor:", float_tensor)
```

Die Funktion `torch.randn()` ist besonders wichtig: Sie erzeugt Zufallszahlen aus einer Normalverteilung mit Mittelwert 0 und Standardabweichung 1. Diese Funktion wird häufig verwendet, um die Gewichte neuronaler Netze zu initialisieren.

Tensoren umformen

Oft müssen Sie die Form eines Tensors ändern, ohne die enthaltenen Daten zu verändern. PyTorch bietet dafür mehrere Methoden:

```
import torch

# Tensor mit 12 Elementen erstellen
original = torch.arange(12)
print("Original:", original)
print("Shape:", original.shape)

# Zu 2D umformen (3 Zeilen, 4 Spalten)
reshaped_2d = original.reshape(3, 4)
print("2D umgeformt:\n", reshaped_2d)

# Zu 3D umformen (2 x 2 x 3)
reshaped_3d = original.reshape(2, 2, 3)
print("3D umgeformt:\n", reshaped_3d)
```

Die Ausgabe zeigt, wie die 12 Elemente neu angeordnet werden:

```
Original: tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
Shape: torch.Size([12])
2D umgeformt:
 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
3D umgeformt:
 tensor([[[ 0,  1,  2],
```

```
         [ 3,  4,  5]],

        [[ 6,  7,  8],
         [ 9, 10, 11]]])
```

Der Unterschied zwischen `reshape()` und `view()` ist wichtig zu verstehen: `view()` erstellt eine neue *Ansicht* derselben Daten im Speicher. Wenn Sie den Original-Tensor ändern, ändert sich auch die View:

```
original = torch.arange(12)
viewed = original.view(4, 3)

# Original ändern
original[0] = 100

# Die View zeigt die Änderung ebenfalls!
print("View nach Änderung:\n", viewed)
```

Die Ausgabe zeigt, dass die Änderung am ersten Element auch in der View sichtbar ist:

```
View nach Änderung:
 tensor([[100,   1,   2],
        [  3,   4,   5],
        [  6,   7,   8],
        [  9,  10,  11]])
```

Weitere nützliche Umformungs-Operationen:

```
# Dimensionen der Größe 1 entfernen
tensor_with_ones = torch.randn(1, 3, 1, 4)
print("Original Shape:", tensor_with_ones.shape)  # [1, 3, 1, 4]

squeezed = tensor_with_ones.squeeze()
print("Nach squeeze:", squeezed.shape)  # [3, 4]

# Dimension der Größe 1 hinzufügen
unsqueezed = squeezed.unsqueeze(0)
print("Nach unsqueeze:", unsqueezed.shape)  # [1, 3, 4]

# Zu 1D flach machen
flattened = tensor_with_ones.flatten()
print("Flattened Shape:", flattened.shape)  # [12]
```

Indexierung und Slicing

Der Zugriff auf Tensor-Elemente funktioniert ähnlich wie bei Python-Listen und NumPy-Arrays:

```
matrix = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Einzelnes Element (Zeile 1, Spalte 2)
print("Element [1,2]:", matrix[1, 2])  # 6

# Erste Zeile
```

```
print("Erste Zeile:", matrix[0, :])  # tensor([1, 2, 3])

# Erste Spalte
print("Erste Spalte:", matrix[:, 0])  # tensor([1, 4, 7])

# Untermatrix (erste 2 Zeilen, alle Spalten)
print("Untermatrix:\n", matrix[0:2, :])

# Negative Indizes (letzte Zeile)
print("Letzte Zeile:", matrix[-1, :])  # tensor([7, 8, 9])

# Boolean-Indexierung
print("Werte > 5:", matrix[matrix > 5])  # tensor([6, 7, 8, 9])
```

Tensoren verbinden und aufteilen

Für das Arbeiten mit Batches (Stapeln von Daten) müssen Sie häufig Tensoren zusammenfügen oder aufteilen:

```
# Zwei Matrizen
a = torch.tensor([[1, 2], [3, 4]])
b = torch.tensor([[5, 6], [7, 8]])

# Entlang Dimension 0 verbinden (untereinander)
cat_dim0 = torch.cat([a, b], dim=0)
print("cat dim=0:\n", cat_dim0)
# tensor([[1, 2],
#         [3, 4],
#         [5, 6],
#         [7, 8]])

# Entlang Dimension 1 verbinden (nebeneinander)
cat_dim1 = torch.cat([a, b], dim=1)
print("cat dim=1:\n", cat_dim1)
# tensor([[1, 2, 5, 6],
#         [3, 4, 7, 8]])

# Stapeln (erstellt neue Dimension)
stacked = torch.stack([a, b])
print("Gestapelt Shape:", stacked.shape)  # torch.Size([2, 2, 2])
print("Gestapelt:\n", stacked)
# tensor([[[1, 2],
#          [3, 4]],
#
#         [[5, 6],
#          [7, 8]]])
```

Der Unterschied zwischen `cat()` und `stack()` ist wichtig: - `cat()` verbindet Tensoren entlang einer *existierenden* Dimension - `stack()` erstellt eine *neue* Dimension und stapelt die Tensoren dort

1.4 Mathematische Operationen

PyTorch bietet umfangreiche Möglichkeiten für mathematische Operationen auf Tensoren. Diese Operationen bilden die Grundlage für alle Berechnungen in neuronalen Netzen.

Elementweise Operationen

Bei elementweisen Operationen wird jedes Element eines Tensors mit dem entsprechenden Element eines anderen Tensors verknüpft:

```
import torch

a = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
b = torch.tensor([[5, 6], [7, 8]], dtype=torch.float32)

print("Tensor a:\n", a)
print("Tensor b:\n", b)

# Grundrechenarten
print("Addition:\n", a + b)
print("Subtraktion:\n", a - b)
print("Multiplikation:\n", a * b)  # Element-weise!
print("Division:\n", a / b)

# Potenz und Wurzel
print("a quadriert:\n", torch.pow(a, 2))
print("Wurzel von a:\n", torch.sqrt(a))
```

Die Ausgaben zeigen die elementweisen Berechnungen:

```
Tensor a:
 tensor([[1., 2.],
        [3., 4.]])
Tensor b:
 tensor([[5., 6.],
        [7., 8.]])
Addition:
 tensor([[ 6.,  8.],
        [10., 12.]])
Subtraktion:
 tensor([[-4., -4.],
        [-4., -4.]])
Multiplikation:
 tensor([[ 5., 12.],
        [21., 32.]])
Division:
 tensor([[0.2000, 0.3333],
        [0.4286, 0.5000]])
a quadriert:
 tensor([[ 1.,  4.],
        [ 9., 16.]])
Wurzel von a:
```

```
 tensor([[1.0000, 1.4142],
        [1.7321, 2.0000]])
```

Beachten Sie, dass `a * b` eine *element-weise* Multiplikation ist – nicht die Matrixmultiplikation aus der linearen Algebra!

Matrixmultiplikation und Batch-Operationen

Die echte Matrixmultiplikation, bei der Zeilen mit Spalten multipliziert und summiert werden, verwenden Sie mit `torch.mm()` oder dem `@`-Operator:

```
# Zwei kompatible Matrizen (2x3 und 3x2)
m1 = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)
m2 = torch.tensor([[7, 8], [9, 10], [11, 12]], dtype=torch.float32)

# Matrixmultiplikation
result = torch.mm(m1, m2)  # Oder: m1 @ m2
print("Matrixmultiplikation:\n", result)
```

Die Ausgabe zeigt das Ergebnis der 2x3 mal 3x2 Matrixmultiplikation, eine 2x2 Matrix:

```
Matrixmultiplikation:
 tensor([[ 58.,  64.],
        [139., 154.]])
```

Das erste Element (58) berechnet sich als: 1×7 + 2×9 + 3×11 = 7 + 18 + 33 = 58.

Batch-Verarbeitung ist ein zentrales Konzept im Deep Learning. Anstatt Daten einzeln zu verarbeiten, werden mehrere Datenpunkte gleichzeitig als "Batch" (Stapel) verarbeitet. Dies ist aus zwei Gründen wichtig: Erstens nutzt es die parallele Rechenkapazität moderner GPUs optimal aus, was das Training erheblich beschleunigt. Zweitens führt die Verarbeitung mehrerer Beispiele pro Update zu stabileren Gradienten.

Für die Verarbeitung mehrerer Matrizen gleichzeitig gibt es `torch.bmm()` (Batch Matrix Multiplication):

```
# Batch von 4 Matrizen der Größe 3x4
batch1 = torch.randn(4, 3, 4)
# Batch von 4 Matrizen der Größe 4x2
batch2 = torch.randn(4, 4, 2)

# Batch-Matrixmultiplikation
result = torch.bmm(batch1, batch2)
print("Batch-Ergebnis Shape:", result.shape)  # [4, 3, 2]
```

Broadcasting-Regeln verstehen

Broadcasting ist ein mächtiges Konzept, das es erlaubt, Operationen zwischen Tensoren unterschiedlicher Größe durchzuführen. PyTorch erweitert dabei automatisch die kleineren Tensoren, um sie kompatibel zu machen. Im folgenden Code wird der Vektor zu jeder Reihe der Matrix addiert:

```python
import torch

vector = torch.tensor([1, 2, 3])
matrix = torch.tensor([[1, 2, 3], [4, 5, 6]])

print("Vektor:", vector)
print("Matrix:\n", matrix)

# Der Vektor wird automatisch auf beide Zeilen angewendet
result = matrix + vector
print("Matrix + Vektor:\n", result)
# tensor([[2, 4, 6],
#         [5, 7, 9]])
```

Die Broadcasting-Regeln sind:

1. **Dimensionen werden von rechts nach links verglichen**: Die Shapes werden rechtsbündig ausgerichtet. Das bedeutet, die letzten Dimensionen werden zuerst verglichen. Bei Shapes `(3, 1, 4)` und `(2, 4)` werden sie so ausgerichtet:

   ```
   (3, 1, 4)
      (2, 4)
   ```

2. **Dimensionen sind kompatibel, wenn**: sie gleich sind, oder eine davon 1 ist
3. **Fehlende Dimensionen werden als 1 behandelt**: Ein Tensor mit Shape `(2, 4)` wird konzeptionell zu `(1, 2, 4)` erweitert

Ein komplexeres Beispiel zeigt, wie die Dimensionen angepasst werden:

```python
a = torch.randn(3, 1, 4)  # Shape: [3, 1, 4]
b = torch.randn(2, 4)     # Shape: [2, 4], wird zu [1, 2, 4]

result = a + b            # Ergebnis-Shape: [3, 2, 4]
print("Broadcasting Ergebnis Shape:", result.shape)
```

Schritt für Schritt passiert Folgendes beim Broadcasting:

1. `b` mit Shape `(2, 4)` wird zu `(1, 2, 4)` erweitert (fehlende Dimension links)
2. Die Shapes werden verglichen:
   - Dim 0: 3 vs 1 → kompatibel, Ergebnis: 3 (die 1 wird auf 3 “gedehnt”, die Werte werden kopiert)
   - Dim 1: 1 vs 2 → kompatibel, Ergebnis: 2 (die 1 wird auf 2 “gedehnt”, die Werte werden kopiert)
   - Dim 2: 4 vs 4 → gleich, Ergebnis: 4
3. Ergebnis-Shape: `(3, 2, 4)`

Statistische und Reduktions-Operationen

Reduktions-Operationen fassen mehrere Werte zu einem zusammen, z.B. wird eine Summe aus bestimmten Werten des Tensors berechnet:

```python
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)

print("Summe aller Elemente:", tensor.sum())        # 21
```

```
print("Mittelwert:", tensor.mean())                # 3.5
print("Maximum:", tensor.max())                    # 6
print("Minimum:", tensor.min())                    # 1
print("Standardabweichung:", tensor.std())

# Reduktion entlang einer bestimmten Dimension
print("Summe pro Zeile (dim=1):", tensor.sum(dim=1))  # tensor([6, 15])
print("Summe pro Spalte (dim=0):", tensor.sum(dim=0)) # tensor([5, 7, 9])
```

Der `dim`-Parameter bestimmt, entlang welcher Dimension reduziert wird. `dim=0` reduziert über die Zeilen (Ergebnis: eine Zeile), `dim=1` reduziert über die Spalten (Ergebnis: eine Spalte).

1.5 Computational Graphs und Gradienten

Jetzt kommen wir zu einem der wichtigsten Konzepte in PyTorch: dem automatischen Berechnen von Gradienten. Diese Fähigkeit ist die Grundlage für das Training neuronaler Netze.

Automatisches Differenzieren verstehen

Erinnern Sie sich noch an Ableitungen aus der Schule? Die Ableitung einer Funktion gibt an, wie stark sich die Ausgabe ändert, wenn sich die Eingabe ein klein wenig ändert. Für die Funktion $f(x) = x^2$ ist die Ableitung $f'(x) = 2x$.

In neuronalen Netzen haben wir Funktionen mit Millionen von Parametern. Die Gradienten sagen uns, in welche Richtung wir jeden Parameter anpassen müssen, um den Fehler zu verringern. PyTorch berechnet diese Gradienten automatisch: ein Prozess, der *automatische Differentiation* oder *Autograd* genannt wird.

Gradienten-Tracking mit `requires_grad`

Um PyTorch mitzuteilen, dass Sie Gradienten für einen Tensor benötigen, setzen Sie `requires_grad=True`:

```
import torch

# Tensoren mit Gradienten-Tracking
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)

print("x:", x)
print("y:", y)
print("x requires grad:", x.requires_grad)

# Operationen auf diesen Tensoren werden verfolgt
z = x * y + x**2
print("z = x * y + x² =", z)
print("z requires grad:", z.requires_grad)

# PyTorch merkt sich, wie z berechnet wurde
print("z grad_fn:", z.grad_fn)
```

Die Ausgabe zeigt die Tensoren und ihre Eigenschaften:

```
x: tensor([2.], requires_grad=True)
y: tensor([3.], requires_grad=True)
x requires grad: True
z = x * y + x² = tensor([10.], grad_fn=<AddBackward0>)
z requires grad: True
z grad_fn: <AddBackward0 object at 0x...>
```

Beachten Sie, dass `z` automatisch auch `requires_grad=True` hat und dass sich PyTorch die letzte Operation (`AddBackward0`) gemerkt hat. Das Ergebnis 10 berechnet sich als: $2 \times 3 + 2^2 = 6 + 4 = 10$.

Sie können Gradienten-Tracking auch nachträglich aktivieren:

```
a = torch.randn(3, 3)
print("requires_grad vorher:", a.requires_grad)  # False

a.requires_grad_(True)  # In-place aktivieren
print("requires_grad nachher:", a.requires_grad)  # True
```

Die `backward()`-Methode

Die Methode `backward()` berechnet die Gradienten für alle Tensoren mit `requires_grad=True`, die zur Berechnung des Ergebnisses beigetragen haben:

```
import torch

# Parameter definieren
x = torch.tensor([2.0], requires_grad=True)

# Berechnung durchführen: y = x² + 3x
y = x**2 + 3*x

# Gradienten berechnen
y.backward()

# Gradient von y bezüglich x: dy/dx = 2x + 3 = 2*2 + 3 = 7
print("Gradient von x:", x.grad)  # tensor([7.])
```

Die `backward()`-Methode *füllt* das `.grad`-Attribut mit den berechneten Ableitungen. Für $y = x^2 + 3x$ ist die Ableitung $dy/dx = 2x + 3$. Bei $x = 2$ ergibt das $2 \cdot 2 + 3 = 7$.

No-Gradient-Kontext für Inferenz

Wenn Sie ein trainiertes Modell nur für Vorhersagen verwenden (Inferenz), benötigen Sie keine Gradienten. Mit `torch.no_grad()` können Sie das Gradienten-Tracking temporär deaktivieren:

```
x = torch.tensor([2.0], requires_grad=True)

# Normale Berechnung mit Tracking
y = x * 2
print("Mit Tracking - requires_grad:", y.requires_grad)  # True
```

```
# Ohne Tracking
with torch.no_grad():
    y = x * 2
    print("Ohne Tracking - requires_grad:", y.requires_grad)  # False
```

Der `no_grad()`-Kontext spart Speicher und beschleunigt Berechnungen, da PyTorch keine Informationen für die Gradientenberechnung speichern muss.

1.6 Lernen mit Gradienten: Backpropagation

In diesem letzten Abschnitt werden wir alles zusammenführen und sehen, wie PyTorch tatsächlich *lernt*. Wir werden verstehen, wie Gradienten durch ein Netzwerk fließen und wie sie verwendet werden, um Parameter anzupassen.

Gradientenfluss in neuronalen Netzen

Ein neuronales Netz ist eine Kette von mathematischen Operationen. Beim Training fließen die Gradienten rückwärts durch diese Kette, von der Ausgabe zur Eingabe. Jede Schicht empfängt Gradienten, berechnet ihre eigenen Gradienten und gibt sie an die vorherige Schicht weiter.

Die Kettenregel aus der Analysis macht dies möglich: Wenn y = f(g(x)), dann ist dy/dx = f'(g(x)) · g'(x). PyTorch wendet diese Regel automatisch auf beliebig komplexe Berechnungen an.

Gradienten-Akkumulation

Ein wichtiges Detail: In PyTorch werden Gradienten standardmäßig *akkumuliert* (addiert), nicht ersetzt. Das bedeutet: Wenn Sie `backward()` mehrfach aufrufen, wird der neue Gradient zum bereits vorhandenen Gradienten hinzuaddiert, anstatt ihn zu überschreiben. Bei "ersetzt" würde jeder `backward()`-Aufruf den alten Gradienten komplett mit dem neuen überschreiben. Bei "akkumuliert" werden alle Gradienten aufsummiert:

```
import torch

x = torch.tensor([2.0], requires_grad=True)

# Erste Berechnung
y1 = x**2
y1.backward()
print("Gradient nach erstem backward:", x.grad)  # tensor([4.])

# Zweite Berechnung (Gradient wird addiert!)
y2 = x**3
y2.backward()
print("Gradient nach zweitem backward:", x.grad)  # tensor([16.])
# Das ist 4 (von vorher) + 12 (neuer Gradient: 3*2² = 12)
```

Wann und warum Gradienten zurücksetzen?

Das Training eines neuronalen Netzes besteht aus vielen *Iterationen* (auch "Trainingsschritte" genannt). Wir brauchen dazu Trainingsdaten mit Eingaben und erwarteten Ausgaben (Vorhersagen). In jeder Iteration passiert dasselbe:

1. Das Netzwerk erhält Eingabedaten und berechnet eine Vorhersage (Forward Pass)
2. Der Fehler (Loss) zwischen Vorhersage und wahrer, erwarteter Ausgabe wird berechnet
3. Die Gradienten werden berechnet (Backward Pass)
4. Die Parameter werden basierend auf den Gradienten angepasst

Da PyTorch Gradienten akkumuliert, müssen Sie vor jeder neuen Iteration die Gradienten zurücksetzen. Andernfalls würden sich die Gradienten aus allen vorherigen Iterationen aufsummieren. Wir benutzen dazu die Tensor-Methode `.zero_()`:

```
if x.grad is not None:
    x.grad.zero_()
```

Loss-Funktionen verstehen

Die *Loss-Funktion* (Verlustfunktion) ist das zentrale Feedback-Signal für das Training. Sie misst, wie weit die Vorhersagen des Modells von den wahren Werten entfernt sind, und komprimiert diesen Unterschied zu einer einzigen Zahl. Ein hoher Loss bedeutet schlechte Vorhersagen, ein niedriger Loss bedeutet gute Vorhersagen. Das Ziel des Trainings ist es, diesen Loss zu minimieren.

Der häufigste Loss für Regressionsprobleme (Vorhersage von kontinuierlichen Werten wie 0.5983) ist der *Mean Squared Error* (MSE):

```
MSE = Mittelwert von (Vorhersage - Wahrheit)²
```

Die Quadrierung hat zwei Vorteile: Erstens werden positive und negative Abweichungen gleich behandelt (beide sind "schlecht"). Zweitens werden große Fehler stärker bestraft als kleine, da $2^2 = 4$, aber $10^2 = 100$.

Wir werden später weitere Loss-Funktionen kennenlernen, zum Beispiel Cross-Entropy für Klassifikationsprobleme.

Der Loss wird immer *am Ende* des Netzwerks berechnet, nachdem alle Schichten ihre Berechnungen durchgeführt haben. Dies ist entscheidend für Backpropagation: Vom Loss aus können wir rückwärts durch alle Operationen gehen und für jeden Parameter berechnen, wie stark er zum Fehler beigetragen hat. Diese Gradienten zeigen uns, in welche Richtung und wie stark wir jeden Parameter anpassen müssen.

Wir passen die Parameter dann in die *entgegengesetzte* Richtung des Gradienten an. Ist der Gradient positiv, verringern wir den Parameter; ist er negativ, erhöhen wir ihn. So bewegen wir uns schrittweise zu einem kleineren Loss. Daher der Name *Gradient Descent* (Gradientenabstieg).

Ein vollständiges Lernbeispiel

Lassen Sie uns all dies in einem konkreten Beispiel zusammenführen. Wir werden ein einfaches lineares Modell trainieren, das die Funktion $y = 2x + 1$ lernt:

```python
import torch

# Trainingsdaten generieren: y = 2x + 1 (mit etwas Rauschen)
torch.manual_seed(42)
x_data = torch.linspace(-2, 2, 20)
y_true = 2 * x_data + 1 + 0.1 * torch.randn_like(x_data)

# Parameter initialisieren (das sind die Werte, die wir lernen wollen)
a = torch.tensor([0.0], requires_grad=True)  # Sollte 2 werden
b = torch.tensor([0.0], requires_grad=True)  # Sollte 1 werden

learning_rate = 0.01

print(f"Startparameter: a={a.item():.3f}, b={b.item():.3f}")

# Trainingsschleife
for epoch in range(100):
    # Gradienten zurücksetzen
    if a.grad is not None:
        a.grad.zero_()
    if b.grad is not None:
        b.grad.zero_()

    # Forward Pass: Vorhersage berechnen
    y_pred = a * x_data + b

    # Loss berechnen (Mean Squared Error)
    loss = torch.mean((y_pred - y_true) ** 2)

    # Backward Pass: Gradienten berechnen
    loss.backward()

    # Parameter aktualisieren (Gradient Descent)
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad

    if epoch % 20 == 0:
        print(f"Epoch {epoch}: Loss={loss.item():.4f}, "
              f"a={a.item():.3f}, b={b.item():.3f}")

print(f"\nEndparameter: a={a.item():.3f}, b={b.item():.3f}")
print("Zielparameter: a=2.000, b=1.000")
```

Wenn Sie dieses Programm ausführen, werden Sie sehen, wie sich die Parameter a und b schrittweise den Zielwerten 2 und 1 annähern. Der Loss sinkt dabei kontinuierlich.

Dieses einfache Beispiel enthält bereits alle wesentlichen Elemente des Trainings neuronaler Netze:

1. **Forward Pass**: Berechnung der Vorhersage aus Eingaben und Parametern
2. **Loss-Berechnung**: Messung des Fehlers zwischen Vorhersage und Wahrheit

3. **Backward Pass**: Berechnung der Gradienten mit `backward()`
4. **Parameter-Update**: Anpassung der Parameter in Richtung des negativen Gradienten

Zusammenfassung

In diesem Kapitel haben Sie die Grundlagen von PyTorch kennengelernt:

- **Tensoren** sind die fundamentale Datenstruktur in PyTorch – mehrdimensionale Arrays, die auf der GPU laufen und Gradienten verfolgen können.
- **Tensor-Operationen** wie Reshaping, Indexierung und mathematische Berechnungen bilden das Handwerkszeug für die Arbeit mit neuronalen Netzen.
- **Autograd** ermöglicht die automatische Berechnung von Gradienten durch Berechnungsgraphen.
- **Gradient Tracking** mit `requires_grad=True` ist die Voraussetzung für das Lernen.
- **Backpropagation** verwendet Gradienten, um Parameter zu optimieren und den Loss zu minimieren.

Diese Konzepte bilden das Fundament für alles, was folgt. Im nächsten Kapitel werden wir darauf aufbauen und unser erstes echtes neuronales Netz implementieren – ein Perzeptron und später ein Multi-Layer Perceptron, das komplexere Probleme lösen kann.

Kapitel 2: Neuronale Netze - Die Grundlagen

Im ersten Kapitel haben Sie gelernt, wie Tensoren in PyTorch funktionieren und wie automatische Differentiation das Berechnen von Gradienten ermöglicht. In diesem Kapitel werden wir diese Grundlagen nutzen, um unser erstes neuronales Netz zu bauen. Wir beginnen mit dem einfachsten Modell – dem Perzeptron – und arbeiten uns bis zu mehrschichtigen Netzwerken vor, die mit echten Datensätzen trainiert werden.

2.1 Das Perzeptron

Was ist ein Perzeptron?

Das Perzeptron ist die einfachste Form eines künstlichen neuronalen Netzes und wurde bereits 1958 von Frank Rosenblatt entwickelt. Trotz seiner Einfachheit bildet es die Grundlage für alle modernen neuronalen Netze.

Ein Perzeptron besteht aus: - **Eingaben** (x_1, x_2, ...): Die Daten, die wir verarbeiten wollen - **Gewichten** (w_1, w_2, ...): Lernbare Parameter, die die Wichtigkeit jeder Eingabe bestimmen - **Bias** (b): Ein zusätzlicher lernbarer Parameter, der als Schwellenwert fungiert - **Aktivierungsfunktion**: Eine Funktion, die das Ergebnis in einen bestimmten Wertebereich transformiert

Die mathematische Berechnung ist überraschend einfach:

`output = activation(w₁*x₁ + w₂*x₂ + ... + b)`

Die Gewichte multipliziert mit den Eingaben werden summiert, der Bias wird addiert, und das Ergebnis wird durch eine Aktivierungsfunktion geleitet. In der Praxis wird oft die Sigmoid-Funktion verwendet, die jeden Wert in den Bereich zwischen 0 und 1 transformiert. Die Sigmoid-Funktion ist definiert als σ(x) = 1 / (1 + e^(-x)). Bei sehr negativen Eingaben nähert sich der Wert 0, bei sehr positiven Eingaben nähert er sich 1, und bei 0 liegt der Wert genau bei 0.5. Diese S-förmige Kurve macht die Funktion ideal für die Interpretation als Wahrscheinlichkeit: Werte nahe 1 bedeuten “ja” (oder “aktiv”), Werte nahe 0 bedeuten “nein” (oder “inaktiv”).

Lineare Trennbarkeit

Ein einzelnes Perzeptron kann nur Probleme lösen, die *linear trennbar* sind. Das bedeutet: Die verschiedenen Klassen (zum Beispiel “Spam” und “Kein Spam”) müssen sich durch eine gerade Linie (oder in höheren Dimensionen durch eine Hyperebene) voneinander trennen lassen.

Stellen Sie sich vor, Sie haben Punkte auf einem Blatt Papier, einige rot und einige grün gefärbt. Wenn Sie eine einzige gerade Linie zeichnen können, die alle roten von allen grünen Punkten trennt, dann ist das Problem linear trennbar.

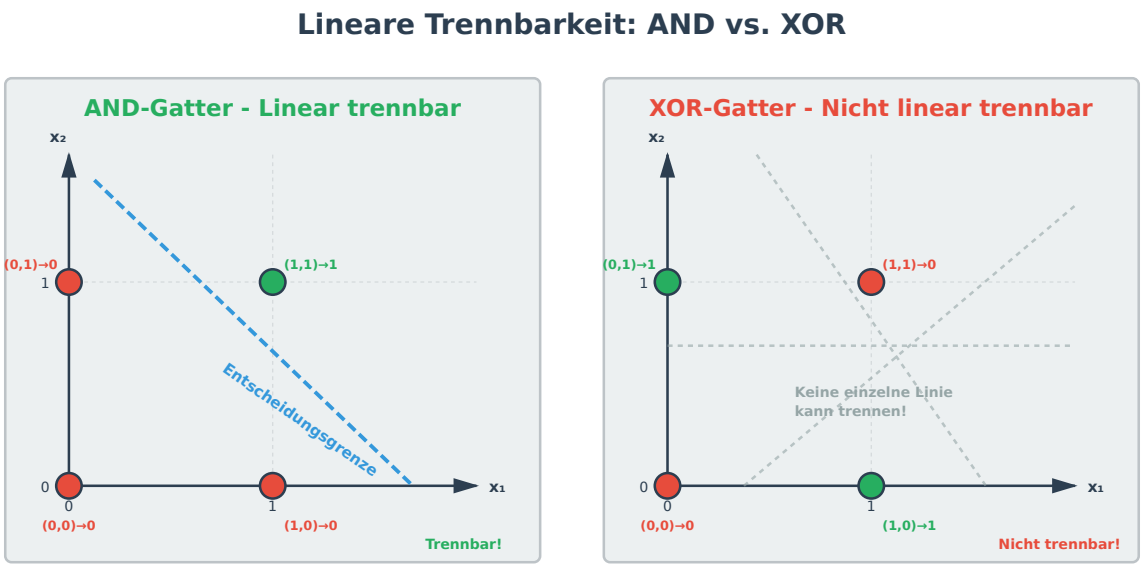

Abbildung 1: Lineare Trennbarkeit: Links ein linear trennbares Problem (AND), rechts ein nicht linear trennbares Problem (XOR)

Ein einfaches Beispiel: Das AND-Gatter

Ein klassisches Beispiel für ein linear trennbares Problem ist das logische AND-Gatter aus der Informatik. Die Wahrheitstabelle sieht so aus:

Eingabe 1	Eingabe 2	Ausgabe
0	0	0
0	1	0
1	0	0
1	1	1

Die Ausgabe ist nur dann 1, wenn *beide* Eingaben 1 sind. Dieses Problem ist linear trennbar: Eine diagonale Linie kann den Punkt (1,1) von den anderen drei Punkten trennen.

Implementieren wir nun ein Perzeptron in PyTorch, das diese Funktion lernt:

```
import torch


class SimplePerceptron:
    def __init__(self, input_size):
        # Gewichte und Bias mit Nullen initialisieren
        self.weights = torch.zeros(input_size, requires_grad=True)
        self.bias = torch.zeros(1, requires_grad=True)

    def forward(self, x):
```

```
        # Lineare Kombination: w*x + b
        linear_output = torch.sum(self.weights * x) + self.bias
        # Sigmoid-Aktivierung anwenden
        return torch.sigmoid(linear_output)
```

Die `forward`-Methode berechnet die Ausgabe des Perzeptrons für eine gegebene Eingabe. Zunächst werden die Gewichte elementweise mit der Eingabe multipliziert und aufsummiert. Dann wird der Bias addiert. Schließlich transformiert die Sigmoid-Funktion das Ergebnis in den Bereich [0, 1].

Die Perzeptron-Lernregel

Das Training eines Perzeptrons folgt einer einfachen Regel: Wenn die Vorhersage falsch ist, passe die Gewichte an. Die Anpassung erfolgt proportional zum Fehler und zur Eingabe:

```
neues_gewicht = altes_gewicht + lernrate * fehler * eingabe
```

Der Fehler ist die Differenz zwischen dem erwarteten Wert (Target) und der Vorhersage. Die Lernrate kontrolliert, wie stark die Gewichte bei jedem Schritt angepasst werden.

Warum multiplizieren wir den Fehler mit der Eingabe? Jede Eingabe trägt unterschiedlich zum Fehler bei. Wenn eine Eingabe groß ist und die Vorhersage falsch, muss das zugehörige Gewicht stärker angepasst werden. Wenn eine Eingabe null ist, hat sie nichts zum Fehler beigetragen und ihr Gewicht sollte nicht verändert werden.

Hier ist die Trainingsschleife:

```
def train_perceptron(perceptron, inputs, targets, learning_rate=0.1, epochs=100):
    for epoch in range(epochs):
        total_loss = 0

        for x, target in zip(inputs, targets):
            # Vorwärtsschritt
            prediction = perceptron.forward(x)

            # Fehler berechnen
            loss = target - prediction
            # loss ist ein Tensor, wir brachen den Float-Wert mit item()
            total_loss += loss.item()

            # Gewichte manuell aktualisieren
            with torch.no_grad():
                perceptron.weights += learning_rate * loss * x
                perceptron.bias += learning_rate * loss

        if epoch % 20 == 0:
            print(f"Epoch {epoch}, Gesamtfehler: {total_loss:.4f}")
```

Nun können wir das Perzeptron auf dem AND-Gatter trainieren:

```
# Trainingsdaten für AND-Gatter
inputs = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
```

```
targets = torch.tensor([0.0, 0.0, 0.0, 1.0])

# Perzeptron erstellen und trainieren
perceptron = SimplePerceptron(2)
print(f"Anfangsgewichte: {perceptron.weights}")
print(f"Anfangsbias: {perceptron.bias}")

train_perceptron(perceptron, inputs, targets, epochs=100)

print(f"\nEndgewichte: {perceptron.weights}")
print(f"Endbias: {perceptron.bias}")

# Testen
print("\nTest des trainierten Perzeptrons:")
for x, target in zip(inputs, targets):
    prediction = perceptron.forward(x)
    print(f"Eingabe: {x.tolist()}, Ziel: {target:.1f}, "
          f"Vorhersage: {prediction.item():.4f}")
```

Nach dem Training sollten Sie sehen, dass das Perzeptron für die Eingabe (1,1) einen Wert nahe 1 vorhersagt und für alle anderen Eingaben Werte nahe 0.

Grenzen des Perzeptrons: Das XOR-Problem

Das AND-Gatter ist ein einfaches Problem. Versuchen wir nun, das XOR-Gatter zu lernen:

Eingabe 1	Eingabe 2	Ausgabe
0	0	0
0	1	1
1	0	1
1	1	0

Schauen Sie sich noch einmal das Bild zur linearen Trennbarkeit oben an: Im rechten Teil sehen Sie das XOR-Problem visualisiert. Die Punkte (0,0) und (1,1) gehören zur Klasse 0, die Punkte (0,1) und (1,0) zur Klasse 1 – keine gerade Linie kann diese beiden Gruppen trennen.

XOR (Exklusiv-Oder) gibt 1 zurück, wenn *genau eine* der Eingaben 1 ist. Wenn Sie die Punkte visualisieren, werden Sie feststellen, dass keine gerade Linie die 0er von den 1er trennen kann. Die Punkte (0,0) und (1,1) ergeben 0, die Punkte (0,1) und (1,0) ergeben 1 – sie liegen diagonal zueinander.

Wenn Sie versuchen, ein einzelnes Perzeptron auf XOR zu trainieren, wird es nie konvergieren:

```
# XOR-Trainingsdaten
targets_xor = torch.tensor([0.0, 1.0, 1.0, 0.0])

perceptron_xor = SimplePerceptron(2)
train_perceptron(perceptron_xor, inputs, targets_xor, epochs=100)
```

```
print("\nTest auf XOR:")
for x, target in zip(inputs, targets_xor):
    prediction = perceptron_xor.forward(x)
    print(f"Eingabe: {x.tolist()}, Ziel: {target:.1f}, "
          f"Vorhersage: {prediction.item():.4f}")
```

Das Perzeptron wird keine korrekten Vorhersagen machen können. Die Vorhersagen werden alle um 0.5 herum liegen – das Netzwerk hat effektiv "aufgegeben", weil es das Problem nicht lösen kann.

Diese Einschränkung führte in den 1960er Jahren zu einer Krise in der KI-Forschung, dem sogenannten "KI-Winter". Die Lösung? Mehrere Schichten von Perzeptronen hintereinander schalten.

2.2 Multi-Layer Perceptrons

Warum mehrere Schichten?

Die Lösung für nicht-linear trennbare Probleme wie XOR ist elegant: Wir stapeln mehrere Schichten von Perzeptronen übereinander. Jede Schicht kann eine neue "Repräsentation" der Daten erzeugen, und die finale Schicht trifft die Entscheidung auf Basis dieser transformierten Daten.

Ein Multi-Layer Perceptron (MLP) besteht aus: - **Eingabeschicht**: Die ursprünglichen Daten - **Versteckte Schichten** (Hidden Layers): Eine oder mehrere Schichten, die Zwischenrepräsentationen berechnen - **Ausgabeschicht**: Die finale Vorhersage

Warum funktioniert das? Denken Sie an das XOR-Problem. Die erste Schicht kann lernen, die Daten so zu transformieren, dass sie in einem neuen Raum linear trennbar werden. Mathematisch ausgedrückt: XOR lässt sich als Kombination von AND und OR darstellen.

Dimensionalität der versteckten Schicht

Die versteckte Schicht muss nicht dieselbe Größe wie die Eingabe haben und genau das macht sie so mächtig. Die Anzahl der Neuronen in der versteckten Schicht (die "Breite" des Layers) ist ein Hyperparameter, den wir selbst wählen.

Für unser XOR-Problem mit 2 Eingaben verwenden wir 4 versteckte Neuronen. Warum gerade 4? Die versteckte Schicht transformiert unsere 2-dimensionalen Eingabepunkte in einen 4-dimensionalen Raum. In diesem höherdimensionalen Raum können die zuvor nicht trennbaren Punkte plötzlich linear trennbar werden. Jedes Neuron in der versteckten Schicht kann ein anderes "Muster" in den Eingabedaten erkennen: Ein Neuron könnte lernen, "beide Eingaben sind hoch" zu erkennen, ein anderes "die erste Eingabe ist hoch", und so weiter.

Die Wahl der richtigen Größe ist eine Abwägung: Zu wenige Neuronen können das Problem nicht ausreichend repräsentieren (Underfitting). Zu viele Neuronen können dazu führen, dass das Netzwerk die Trainingsdaten auswendig lernt statt zu generalisieren (Overfitting). Für einfache Probleme wie XOR reichen 4 Neuronen völlig aus. In der Praxis experimentiert man oft mit verschiedenen Größen und wählt die kleinste, die gute Ergebnisse liefert.

Matrix-Operationen im Batch-Modus

Im ersten Kapitel haben Sie bereits Matrixmultiplikation kennengelernt. In neuronalen Netzen verwenden wir Matrixoperationen, um viele Berechnungen gleichzeitig durchzuführen.

Statt jede Eingabe einzeln zu verarbeiten, fassen wir mehrere Eingaben zu einem *Batch* zusammen. Die Gewichte einer Schicht werden als Matrix organisiert:

```
[batch_size, input_features] @ [input_features, hidden_features]
    = [batch_size, hidden_features]
```

Hier ein konkretes Beispiel: Angenommen, wir haben 2 Eingaben und 3 versteckte Neuronen. Unsere Gewichtsmatrix W hat die Form [2, 3] und ein Batch mit 4 Beispielen hat die Form [4, 2]:

```
Batch X (4 Beispiele, 2 Features):   Gewichte W (2 Eingabefeatures, 3 Ausgabeneuronen):
[[0, 0],                                 [[0.5, 0.3, 0.1],
 [0, 1],                   @              [0.2, 0.4, 0.6]]
 [1, 0],
 [1, 1]]

= Ergebnis (4 Beispiele, 3 Features):
[[0.0, 0.0, 0.0],     # 0*0.5 + 0*0.2, 0*0.3 + 0*0.4, 0*0.1 + 0*0.6
 [0.2, 0.4, 0.6],     # 0*0.5 + 1*0.2, 0*0.3 + 1*0.4, 0*0.1 + 1*0.6
 [0.5, 0.3, 0.1],     # 1*0.5 + 0*0.2, 1*0.3 + 0*0.4, 1*0.1 + 0*0.6
 [0.7, 0.7, 0.7]]     # 1*0.5 + 1*0.2, 1*0.3 + 1*0.4, 1*0.1 + 1*0.6
```

Jede Zeile im Ergebnis entspricht einem Trainingsbeispiel, jede Spalte einem versteckten Neuron. Durch diese Batch-Verarbeitung berechnen wir alle Ausgaben in einer einzigen Matrix-Operation – viel effizienter als eine Schleife über einzelne Beispiele.

Hier ist eine MLP-Implementierung, die Batch-Verarbeitung unterstützt:

```python
import torch

torch.manual_seed(42)


class MultiLayerPerceptron:
    def __init__(self, input_size, hidden_size, output_size):
        # Versteckte Schicht: Gewichte und Bias
        self.W1 = torch.randn(input_size, hidden_size, requires_grad=True)
        self.b1 = torch.randn(hidden_size, requires_grad=True)

        # Ausgabeschicht: Gewichte und Bias
        self.W2 = torch.randn(hidden_size, output_size, requires_grad=True)
        self.b2 = torch.randn(output_size, requires_grad=True)

    def forward(self, x):
        # Versteckte Schicht: z1 = x @ W1 + b1
        z1 = torch.matmul(x, self.W1) + self.b1
        a1 = torch.sigmoid(z1)  # Aktivierung anwenden
```

```
        # Ausgabeschicht: z2 = a1 @ W2 + b2
        z2 = torch.matmul(a1, self.W2) + self.b2
        a2 = torch.sigmoid(z2)  # Aktivierung anwenden

        # Zwischenwerte für Backpropagation speichern
        self.z1, self.a1, self.z2, self.a2 = z1, a1, z2, a2
        return a2
```

Beachten Sie die Zeile `torch.manual_seed(42)` am Anfang des Codes. Der "Seed" initialisiert den Zufallszahlengenerator mit einem festen Startwert. Dadurch erzeugt `torch.randn()` bei jedem Durchlauf dieselben "zufälligen" Zahlen. Das ist wichtig für die Reproduzierbarkeit: Wenn Sie den Code ausführen, erhalten Sie jedesmal exakt dieselben Ergebnisse bzw. können Änderungen im Code vergleichen. In der Praxis verwendet man Seeds während der Entwicklung und entfernt sie oft für die finale Anwendung.

Beachten Sie außerdem, dass wir die Gewichte mit `torch.randn()` initialisieren statt mit Nullen. Bei mehrschichtigen Netzen ist eine zufällige Initialisierung wichtig: Wenn alle Gewichte gleich wären, würden alle Neuronen dasselbe lernen (das sogenannte "Symmetrie-Problem").

Lassen Sie uns die Dimensionen verstehen:

```
# MLP für XOR erstellen (2 Eingaben, 4 versteckte Neuronen, 1 Ausgabe)
mlp = MultiLayerPerceptron(input_size=2, hidden_size=4, output_size=1)

print("Gewichte versteckte Schicht:", mlp.W1.shape)  # [2, 4]
print("Bias versteckte Schicht:", mlp.b1.shape)       # [4]
print("Gewichte Ausgabeschicht:", mlp.W2.shape)       # [4, 1]
print("Bias Ausgabeschicht:", mlp.b2.shape)           # [1]

# Batch-Verarbeitung: alle 4 XOR-Eingaben gleichzeitig
batch_input = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
batch_output = mlp.forward(batch_input)

print(f"\nBatch-Eingabe Shape: {batch_input.shape}")   # [4, 2]
print(f"Batch-Ausgabe Shape: {batch_output.shape}")    # [4, 1]
print(f"Ausgaben:\n{batch_output}")
```

Die versteckte Schicht transformiert die 2 Eingabefeatures in 4 "versteckte" Features. Die Ausgabeschicht reduziert diese wieder auf 1 Ausgabewert.

Aktivierungsfunktionen: Sigmoid, ReLU, Tanh

Die Aktivierungsfunktion ist entscheidend dafür, dass das Netzwerk nicht-lineare Zusammenhänge lernen kann. Ohne Aktivierungsfunktion wäre ein tiefes Netz mathematisch äquivalent zu einem einzigen Layer, egal wie viele Schichten Sie stapeln.

Die drei wichtigsten Aktivierungsfunktionen sind:

Sigmoid: Transformiert jeden Wert in den Bereich (0, 1)

```
sigmoid(x) = 1 / (1 + e^(-x))
```

Sigmoid war historisch die erste Wahl, hat aber Nachteile: Bei sehr großen oder kleinen Eingaben ist die Ableitung fast null, was das Lernen verlangsamt ("Vanishing Gradient Problem").

Tanh: Transformiert jeden Wert in den Bereich (-1, 1)

```
tanh(x) = (e^x - e^(-x)) / (e^x + e^(-x))
```

Tanh ist "zentriert" um null, was das Training manchmal verbessert, leidet aber unter ähnlichen Problemen wie Sigmoid.

ReLU (Rectified Linear Unit): Die heute am häufigsten verwendete Aktivierung

```
relu(x) = max(0, x)
```

ReLU ist extrem einfach: negative Werte werden zu null, positive bleiben unverändert. Diese Einfachheit macht ReLU schnell zu berechnen und vermeidet weitgehend das Vanishing-Gradient-Problem.

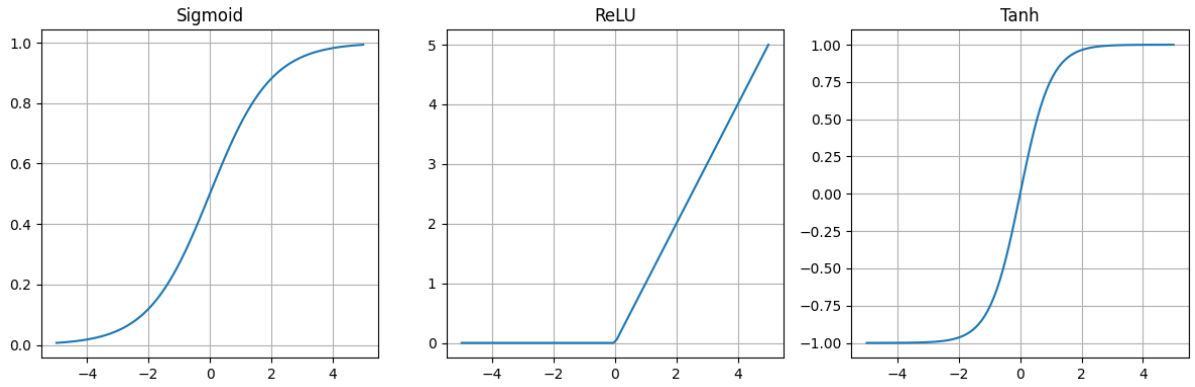

Abbildung 2: Vergleich der drei Aktivierungsfunktionen: Sigmoid (links), Tanh (Mitte), ReLU (rechts)

In PyTorch können Sie diese Funktionen direkt aufrufen:

```
x = torch.tensor([-2.0, -1.0, 0.0, 1.0, 2.0])

print("Sigmoid:", torch.sigmoid(x))
print("Tanh:", torch.tanh(x))
print("ReLU:", torch.relu(x))
```

Die Ausgabe zeigt die unterschiedlichen Bereiche:

```
Sigmoid: tensor([0.1192, 0.2689, 0.5000, 0.7311, 0.8808])
Tanh: tensor([-0.9640, -0.7616,  0.0000,  0.7616,  0.9640])
ReLU: tensor([0., 0., 0., 1., 2.])
```

In der Praxis verwendet man heute meist ReLU in den versteckten Schichten. Die Ausgabeschicht hängt vom Problem ab: Sigmoid für binäre Klassifikation (Ja/Nein), Softmax für Multi-Klassen-Klassifikation, keine Aktivierung für Regression.

Das XOR-Problem lösen

Mit unserem MLP können wir nun das XOR-Problem lösen, das ein einzelnes Perzeptron nicht bewältigen konnte. Das Training werden wir im nächsten Abschnitt

mit PyTorchs automatischer Differenzierung implementieren, aber konzeptionell passiert Folgendes:

Die versteckte Schicht lernt, die vier Eingabepunkte so zu transformieren, dass sie in einem 4-dimensionalen Raum linear trennbar werden. Ein Neuron könnte zum Beispiel lernen, "beide Eingaben sind 1" zu erkennen, ein anderes "mindestens eine Eingabe ist 1". Die Ausgabeschicht kombiniert dann diese Zwischenrepräsentationen zur finalen XOR-Vorhersage.

2.3 Backpropagation und Gradient Descent

Backpropagation verstehen

Backpropagation ("Rückwärtspropagierung") ist der Algorithmus, der das Training neuronaler Netze ermöglicht. Die Grundidee: Wir berechnen, wie stark jedes Gewicht zum Gesamtfehler beigetragen hat, und passen es entsprechend an.

Der Prozess läuft in zwei Phasen ab:

1. **Forward Pass**: Die Eingabe wird durch das Netzwerk geleitet, Schicht für Schicht, bis wir eine Vorhersage erhalten. So haben wir das in den vorhergehenden Abschnitten bereits gemacht.
2. **Backward Pass**: Ausgehend vom Fehler (Loss) berechnen wir die Gradienten für jedes Gewicht, indem wir die Kettenregel der Differentiation rückwärts durch das Netzwerk anwenden.

Die Kettenregel besagt: Wenn y = f(g(x)), dann ist dy/dx = f'(g(x)) · g'(x). In einem neuronalen Netz mit vielen Schichten wird diese Regel wiederholt angewendet: der Gradient "fließt" von der Ausgabe zurück zur Eingabe, wobei er an jeder Schicht mit dem lokalen Gradienten multipliziert wird.

Für den Loss verwenden wir oft den Mean Squared Error:

`Loss = 0.5 * (output - target)²`

Der Faktor 0.5 ist eine Konvention, die die Ableitung vereinfacht: Die Ableitung ist dann einfach (output - target).

Automatische Differentiation mit PyTorch

Im ersten Kapitel haben Sie bereits `backward()` kennengelernt. Jetzt setzen wir es ein, um ein MLP zu trainieren:

```
import torch

torch.manual_seed(42)


class MultiLayerPerceptron:
    def __init__(self, input_size, hidden_size, output_size):
        self.W1 = torch.randn(input_size, hidden_size, requires_grad=True)
        self.b1 = torch.randn(hidden_size, requires_grad=True)
        self.W2 = torch.randn(hidden_size, output_size, requires_grad=True)
        self.b2 = torch.randn(output_size, requires_grad=True)
```

```
    def forward(self, x):
        z1 = torch.matmul(x, self.W1) + self.b1
        a1 = torch.sigmoid(z1)
        z2 = torch.matmul(a1, self.W2) + self.b2
        a2 = torch.sigmoid(z2)
        return a2


# XOR-Daten
xor_inputs = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
xor_targets = torch.tensor([[0.0], [1.0], [1.0], [0.0]])


def train_mlp(mlp, inputs, targets, learning_rate=1.0, epochs=1000):
    losses = []

    for epoch in range(epochs):
        # Forward Pass für den gesamten Batch
        outputs = mlp.forward(inputs)

        # Loss berechnen (Mean Squared Error)
        loss = torch.mean(0.5 * (outputs - targets) ** 2)
        losses.append(loss.item())

        # Backward Pass
        loss.backward()

        # Parameter aktualisieren
        with torch.no_grad():
            mlp.W1 -= learning_rate * mlp.W1.grad
            mlp.b1 -= learning_rate * mlp.b1.grad
            mlp.W2 -= learning_rate * mlp.W2.grad
            mlp.b2 -= learning_rate * mlp.b2.grad

        # Gradienten zurücksetzen
        mlp.W1.grad.zero_()
        mlp.b1.grad.zero_()
        mlp.W2.grad.zero_()
        mlp.b2.grad.zero_()

        if epoch % 200 == 0:
            print(f"Epoch {epoch}, Loss: {loss:.6f}")

    return losses


# Training
mlp = MultiLayerPerceptron(2, 4, 1)
losses = train_mlp(mlp, xor_inputs, xor_targets)

# Testen
```

```
print("\nTest des trainierten MLP auf XOR:")
for x, target in zip(xor_inputs, xor_targets):
    pred = mlp.forward(x.unsqueeze(0))
    print(f"Eingabe: {x.tolist()}, Ziel: {target.item():.1f}, "
          f"Vorhersage: {pred.item():.4f}")
```

Wenn Sie diesen Code ausführen, sollten Sie sehen, dass der Loss kontinuierlich sinkt und die finalen Vorhersagen nahe an den Zielwerten liegen. Das MLP hat das XOR-Problem gelöst!

Gradient Descent Varianten: Batch, Stochastic, Mini-Batch

Es gibt verschiedene Strategien, wie wir die Gradienten berechnen und die Gewichte aktualisieren:

Batch Gradient Descent: Wir berechnen den Gradienten über den *gesamten* Datensatz, bevor wir die Gewichte aktualisieren. Das ist stabil, aber bei großen Datensätzen langsam und speicherintensiv. So passiert das im Beispiel oben.

Stochastic Gradient Descent (SGD): Wir aktualisieren die Gewichte nach *jedem einzelnen* Trainingsbeispiel. Das ist schneller und kann lokalen Minima entkommen, aber die Updates sind "verrauscht".

Mini-Batch Gradient Descent: Ein Kompromiss: Wir verwenden kleine Gruppen (Batches) von typischerweise 16 bis 256 Beispielen. Das kombiniert die Vorteile beider Ansätze und ist heute die Standardmethode.

Hier ist eine Implementation für Stochastic Gradient Descent, wir fügen dem obigen Code ein einfach eine zusätliche for-Schleife über die Input-Daten hinzu:

```
def train_mlp_sgd(mlp, inputs, targets, learning_rate=1.0, epochs=1000):
    """Stochastischer Gradientenabstieg"""
    losses = []

    for epoch in range(epochs):
        total_loss = 0

        for x, target in zip(inputs, targets):
            # Forward Pass für einzelnes Beispiel
            output = mlp.forward(x.unsqueeze(0))

            # Loss berechnen
            loss = 0.5 * (output - target) ** 2
            total_loss += loss.item()

            # Backward Pass
            loss.backward()

            # Parameter aktualisieren
            with torch.no_grad():
                mlp.W1 -= learning_rate * mlp.W1.grad
                mlp.b1 -= learning_rate * mlp.b1.grad
                mlp.W2 -= learning_rate * mlp.W2.grad
                mlp.b2 -= learning_rate * mlp.b2.grad
```

```
            # Gradienten zurücksetzen
            mlp.W1.grad.zero_()
            mlp.b1.grad.zero_()
            mlp.W2.grad.zero_()
            mlp.b2.grad.zero_()

        losses.append(total_loss)

        if epoch % 200 == 0:
            print(f"SGD - Epoch {epoch}, Loss: {total_loss:.6f}")

    return losses
```

Epochs und Steps

Zwei wichtige Begriffe beim Training:

- **Epoch** (Epoche): Ein vollständiger Durchlauf durch den gesamten Trainingsdatensatz
- **Step** (oder Schritt, Iteration): Eine einzelne Gewichtsaktualisierung

Die Anzahl der Steps pro Epoch hängt von der Batch-Größe ab:

`Steps pro Epoch = Anzahl Trainingsbeispiele / Batch-Größe`

Beispiel: Bei 1000 Trainingsbildern und einer Batch-Größe von 100 haben Sie 10 Steps pro Epoch.

In unserem XOR-Beispiel haben wir nur 4 Trainingsbeispiele. Bei Batch Gradient Descent ist ein Step gleich einer Epoch. Bei SGD haben wir 4 Steps pro Epoch.

2.4 Das torch.nn Modul

PyTorchs Neural Network Module

Bisher haben wir alles “von Hand” implementiert: Gewichte, Forward Pass, Gradientenberechnung. Das war lehrreich, aber für größere Projekte unpraktisch. PyTorch bietet mit dem `torch.nn` Modul eine elegante Abstraktion.

Die wichtigsten Bausteine: - `nn.Module`: Die Basisklasse für alle neuronalen Netze - `nn.Linear`: Eine vollständig verbundene Schicht (Dense Layer) mit linearer Abbildung per Matrix-Multiplikation - Verschiedene Aktivierungsfunktionen wie `nn.Sigmoid`, `nn.ReLU` - Loss-Funktionen wie `nn.MSELoss`, `nn.CrossEntropyLoss`

`nn.Linear` und `nn.Module`

Hier ist unser MLP ohne manuelle Matrix-Berechnungen, neu implementiert mit `torch.nn`:

```
import torch
import torch.nn as nn

torch.manual_seed(42)
```

```python
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden = nn.Linear(input_size, hidden_size)
        self.output = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.sigmoid(x)
        return x


# Modell erstellen
model = SimpleNN(input_size=2, hidden_size=4, output_size=1)
print(model)
```

Die Ausgabe zeigt die Struktur des Modells:

```
SimpleNN(
  (hidden): Linear(in_features=2, out_features=4, bias=True)
  (output): Linear(in_features=4, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

Beachten Sie, wie viel kürzer und übersichtlicher der Code geworden ist. `nn.Linear` kümmert sich automatisch um: - Initialisierung der Gewichte und Biases - Korrekte Dimensionen der Matrixmultiplikation - Registrierung der Parameter für `backward()`

Sie können alle Parameter eines Modells auflisten:

```python
print("\nModellparameter:")
for name, param in model.named_parameters():
    print(f"{name}: {param.shape}")
```

Regression vs. Klassifikation

Neuronale Netze können für zwei Hauptaufgaben eingesetzt werden:

Regression: Vorhersage kontinuierlicher Werte (z.B, 0.3254) - Beispiele: Hauspreise, Temperatur, Aktienkurse - Ausgabeschicht: Keine Aktivierung (oder lineare Aktivierung) - Loss: Mean Squared Error (MSE)

Klassifikation: Vorhersage von Kategorien (z.B. “Spam” vs. “Kein Spam”) - Beispiele: Spam-Erkennung (2 Klassen), Bilderkennung (viele Klassen, z.B. “Katze”, “Hund”, “Pferd”, ...) - Ausgabeschicht: Sigmoid (2 Klassen) oder Softmax (mehrere Klassen) - Loss: Binary Cross-Entropy oder Cross-Entropy

Unser XOR-Problem ist ein Klassifikationsproblem mit 2 Klassen (0 oder 1). Deshalb verwenden wir Sigmoid in der Ausgabeschicht.

Loss-Funktionen und Optimizer

Loss-Funktionen haben wir bereits kennengelernt. Sie messen, wie weit unsere Vorhersagen vom Zielwert entfernt sind. Aber wie genau werden die Gewichte angepasst, um den Loss zu minimieren? Hier kommen **Optimizer** ins Spiel.

Ein Optimizer ist ein Algorithmus, der die Gradienten nimmt und daraus berechnet, wie die Gewichte aktualisiert werden sollen. Der einfachste Optimizer ist Stochastic Gradient Descent (SGD), den wir bereits manuell implementiert haben: `gewicht_neu = gewicht_alt - lernrate * gradient`. Aber es gibt fortgeschrittenere Optimizer wie Adam, RMSprop oder AdaGrad, die zusätzliche Techniken verwenden, zum Beispiel adaptive Lernraten für verschiedene Parameter oder "Momentum", das die Richtung vorheriger Updates berücksichtigt. Diese Techniken können das Training beschleunigen und stabilisieren.

PyTorch bietet fertige Implementierungen für Loss-Funktionen und Optimierer:

```
import torch.optim as optim

# Loss-Funktion
loss_fn = nn.MSELoss()

# Optimizer (Stochastic Gradient Descent)
optimizer = optim.SGD(model.parameters(), lr=1.0)
```

Der Optimizer übernimmt die Gewichtsaktualisierung. Sie müssen nicht mehr manuell durch alle Parameter iterieren. Hier ist die vereinfachte Trainingsschleife:

```
def train_with_optimizer(model, inputs, targets, learning_rate=1.0, epochs=2000):
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    losses = []

    for epoch in range(epochs):
        # Forward Pass
        outputs = model(inputs)
        loss = loss_fn(outputs, targets)

        # Backward Pass
        optimizer.zero_grad()  # Gradienten zurücksetzen
        loss.backward()        # Gradienten berechnen
        optimizer.step()       # Parameter aktualisieren

        losses.append(loss.item())

        if epoch % 200 == 0:
            print(f"Epoch {epoch}, Loss: {loss:.6f}")

    return losses
```

Der dreischrittige Prozess `zero_grad()` → `backward()` → `step()` ist das Standardmuster für Training in PyTorch.

Trainieren wir unser neues Modell:

```
xor_inputs = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
xor_targets = torch.tensor([[0.0], [1.0], [1.0], [0.0]])

model = SimpleNN(input_size=2, hidden_size=4, output_size=1)
losses = train_with_optimizer(model, xor_inputs, xor_targets)

# Ergebnisse anzeigen
print("\nTest des trainierten Modells:")
with torch.no_grad():
    for x, target in zip(xor_inputs, xor_targets):
        pred = model(x.unsqueeze(0))
        print(f"Eingabe: {x.tolist()}, Ziel: {target.item():.1f}, "
              f"Vorhersage: {pred.item():.4f}")
```

Wie Sie sehen, lernt das Netzwerk das XOR-Problem genauso erfolgreich wie unsere manuelle Implementierung, aber der Code ist deutlich kürzer und übersichtlicher.

2.5 Trainierte Modelle anwenden

Training vs. Evaluation Mode

Nach dem Training wollen wir die Modell für neue Daten einsetzen, z.B. wollen wir unsere trainierten Deep-Learning-Modelle später in eine Anwendung einbauen. Das Modell wird dann vom Anwender immer wieder auf neue Daten angewendet. Dann soll das Modell aber nicht mehr lernen, d.h. alle Parameter behalten ihre trainierten Werte und wir müssen keine Gradienten mehr berechnen. Wir müssen insgesamt weniger Berechnungen durchführen und die Ausgabe erfolgt schneller, bei großen Modellen sogar deutlich schneller.

PyTorch-Modelle haben dazu zwei Modi:

- **Training Mode** (`model.train()`): Einige Schichten verhalten sich anders, zum Beispiel ist Dropout aktiv (dazu später mehr)
- **Evaluation Mode** (`model.eval()`): Das Modell ist für Vorhersagen optimiert

Für Inferenz (Vorhersagen auf neuen Daten) sollten Sie immer:

```
model.eval()  # Evaluationsmodus aktivieren
```

`torch.no_grad()` für effiziente Inferenz

Bei Vorhersagen brauchen wir keine Gradienten. Mit `torch.no_grad()` sparen wir Speicher und beschleunigen die Berechnung:

```
def evaluate_model(model, inputs, targets):
    model.eval()

    with torch.no_grad():
        outputs = model(inputs)

        # Mean Squared Error berechnen
        mse = nn.MSELoss()(outputs, targets)
```

```
        # Accuracy für binäre Klassifikation
        predictions = (outputs > 0.5).float()
        accuracy = (predictions == targets).float().mean()

        print(f"MSE: {mse:.6f}")
        print(f"Accuracy: {accuracy:.4f}")

        return outputs, predictions


# Auswertung
outputs, predictions = evaluate_model(model, xor_inputs, xor_targets)
```

Metriken: Accuracy, Precision, Recall

Für Klassifikationsprobleme gibt es verschiedene Metriken:

Accuracy (Genauigkeit): Anteil der korrekten Vorhersagen

```
Accuracy = Richtige Vorhersagen / Alle Vorhersagen
```

Precision (Präzision): Von allen positiv vorhergesagten, wie viele sind tatsächlich positiv?

```
Precision = True Positives / (True Positives + False Positives)
```

Recall (Sensitivität): Von allen tatsächlich positiven, wie viele wurden als positiv erkannt?

```
Recall = True Positives / (True Positives + False Negatives)
```

Bei unserem XOR-Beispiel mit nur 4 Datenpunkten sind diese Metriken weniger aussagekräftig. Im nächsten Abschnitt arbeiten wir mit einem echten Datensatz.

Lernkurven visualisieren

Lernkurven zeigen, wie sich der Loss über die Trainingszeit entwickelt. Sie helfen zu erkennen, ob das Modell lernt und wann das Training abgeschlossen ist:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 4))
plt.plot(losses)
plt.title("Lernkurve: XOR-Problem")
plt.xlabel("Epoch")
plt.ylabel("Loss (MSE)")
plt.grid(True)
plt.tight_layout()
plt.savefig("lernkurve.png", dpi=100)
plt.show()
```

Eine gute Lernkurve fällt schnell am Anfang und flacht dann ab. Wenn der Loss nicht sinkt, ist die Lernrate möglicherweise zu klein. Wenn er stark schwankt, ist sie möglicherweise zu groß.

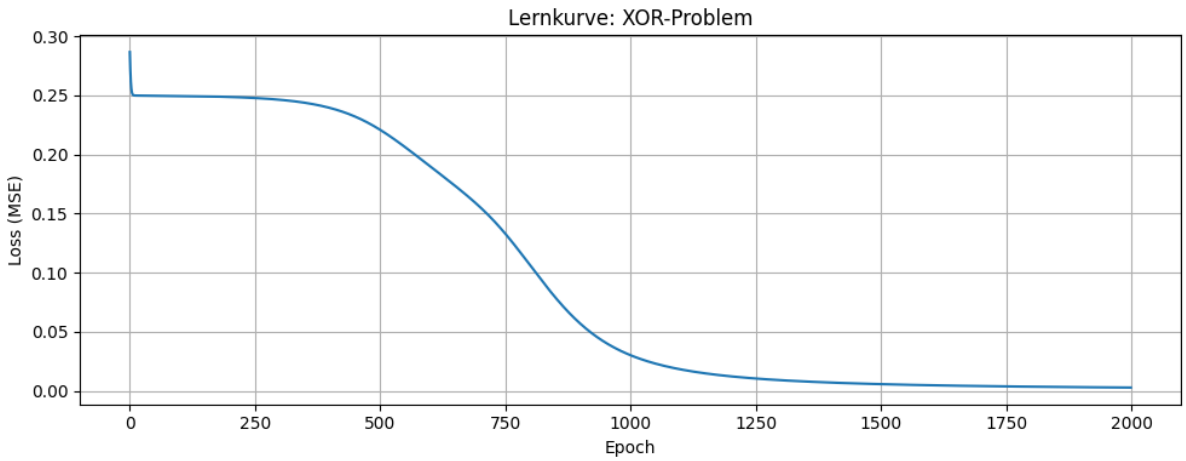

Abbildung 3: Lernkurve für das XOR-Problem: Der Loss sinkt schnell in den ersten Epochen und nähert sich dann asymptotisch einem niedrigen Wert

2.6 Multi-Klassen-Klassifikation mit Datensätzen

Der Wine-Datensatz

Für ein realistischeres Beispiel verwenden wir den Wine-Datensatz aus der Python-Bibliothek scikit-learn. Er enthält chemische Analysen von 178 italienischen Weinen aus drei verschiedenen Anbaugebieten. Anhand von 13 chemischen Eigenschaften (Alkoholgehalt, Apfelsäure, Asche, etc.) soll das Modell vorhersagen, aus welchem Anbaugebiet ein Wein stammt.

```
from sklearn.datasets import load_wine

wine = load_wine()
X, y = wine.data, wine.target

print(f"Datensatz: {X.shape[0]} Proben, {X.shape[1]} Features")
print(f"Klassen: {wine.target_names}")
print(f"Einige Features: {wine.feature_names[:5]}...")
```

Datensätze in Trainings- und Validierungsdaten aufteilen

Beim maschinellen Lernen ist es entscheidend, die Daten aufzuteilen:

- **Trainingsdaten**: Zum Lernen der Muster
- **Validierungsdaten**: Zum Überwachen des Trainingsfortschritts
- **Testdaten**: Zur finalen Evaluation (unberührt während der Entwicklung)

Warum brauchen wir sowohl Validierungs- als auch Testdaten? Die Validierungsdaten nutzen wir während der Entwicklung, um Hyperparameter (wie Lernrate, Netzwerkgröße, Anzahl der Epochen) zu optimieren. Wenn das Modell auf den Validierungsdaten gut funktioniert, wählen wir diese Konfiguration. Aber dadurch "passt" sich unser Modell indirekt auch an die Validierungsdaten an. Die Testdaten bleiben komplett unberührt, bis wir mit der Entwicklung fertig sind. Sie geben uns eine ehrliche Einschätzung, wie gut das Modell auf völlig neuen Daten funktioniert. Für unser einfaches Beispiel verzichten wir auf ein separates Testset und verwenden nur die Aufteilung in Training und Validierung.

Wir verwenden die Funktion `train_test_split()` aus scikit-learn für die Aufteilung:

```
from sklearn.model_selection import train_test_split

# 80% Training, 20% Validierung
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

print(f"Trainingsset: {X_train.shape}")
print(f"Validierungsset: {X_val.shape}")
```

Die Parameter der Funktion im Detail: `test_size=0.2` bedeutet, dass 20% der Daten für die Validierung reserviert werden (der Name "test_size" ist historisch bedingt). Der Parameter `random_state=42` ist ein Seed für den Zufallszahlengenerator, der die Aufteilung reproduzierbar macht, bei gleichem Seed erhalten Sie immer dieselbe Aufteilung. Der Parameter `stratify=y` sorgt dafür, dass die Klassenverteilung in beiden Sets gleich bleibt. Wenn im Gesamtdatensatz 40% der Beispiele zur Klasse 0 gehören, sind es auch in Training und Validierung jeweils 40%. Das ist wichtig bei unbalancierten Datensätzen, damit nicht zufällig alle Beispiele einer seltenen Klasse im Validierungsset landen.

Warum normalisieren?

Schauen wir uns die Wertebereiche der Features an: - Alkohol: 11 - 15 - Prolin: 278 - 1680

Diese unterschiedlichen Skalen können Probleme verursachen. Features mit größeren Werten dominieren das Training, obwohl sie nicht unbedingt wichtiger sind. Die Lösung: eine Skalierung der Daten.

Es gibt zwei verwandte, aber unterschiedliche Techniken: **Standardisierung** transformiert die Daten so, dass sie Mittelwert 0 und Standardabweichung 1 haben. **Normalisierung** (auch Min-Max-Skalierung genannt) transformiert die Daten in einen festen Bereich, typischerweise [0, 1]. Im alltäglichen Sprachgebrauch werden beide Begriffe oft synonym verwendet. Wenn jemand von "Normalisierung" spricht, meint er oft Standardisierung. Für neuronale Netze funktionieren beide Methoden gut, aber Standardisierung ist die häufigere Wahl, weil sie robuster gegenüber Ausreißern ist.

Standardisierung transformiert jedes Feature, sodass es Mittelwert 0 und Standardabweichung 1 hat:

```
x_standardisiert = (x - Mittelwert) / Standardabweichung
```

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)  # Lernt und transformiert
X_val_scaled = scaler.transform(X_val)          # Nur transformieren!
```

Wichtig: Der Scaler wird nur auf den Trainingsdaten "gefittet". Die Validierungsdaten werden mit denselben Parametern (Mittelwert und Standardabweichung aus

den Trainingsdaten) transformiert. Sonst würde Information aus den Validierungsdaten in das Training "lecken" (engl. "leak").

DataLoaders und Batching

In unseren bisherigen Beispielen hatten wir nur 4 Datenpunkte (XOR). In der Praxis arbeiten wir mit Tausenden oder Millionen von Beispielen. Es wäre ineffizient, und bei großen Datensätzen unmöglich, alle Daten gleichzeitig durch das Netzwerk zu schicken. Stattdessen verarbeiten wir die Daten in kleinen Gruppen (Batches).

PyTorch bietet dafür zwei zusammenarbeitende Konzepte: **Dataset** speichert die Daten und ermöglicht den Zugriff auf einzelne Beispiele über einen Index. **DataLoader** nimmt ein Dataset und liefert Batches von Beispielen. Er kümmert sich um das Aufteilen in Batches, das zufällige Mischen der Daten, und kann sogar mehrere Batches parallel im Hintergrund vorbereiten, während das Netzwerk trainiert.

Das einfachste Dataset ist `TensorDataset`, das mehrere Tensoren (Features und Labels) zusammenfasst. Der DataLoader iteriert dann über diese Paare in konfigurierbaren Batch-Größen:

PyTorch bietet `DataLoader` für effizientes Batching und Shuffling:

```
import torch
from torch.utils.data import DataLoader, TensorDataset

# Zu PyTorch-Tensoren konvertieren
X_train_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
X_val_tensor = torch.tensor(X_val_scaled, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train)
y_val_tensor = torch.tensor(y_val)

# Datasets erstellen
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)

# DataLoader erstellen
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

Wie oben beschrieben kümmert sich der DataLoader im Beispiel um: - Aufteilung in Batches der angegebenen Größe - Shuffling der Daten (wichtig für Training, damit das Modell nicht die Reihenfolge lernt) - Effizientes Laden der Daten

Sie können dann per DataLoader über die Batches iterieren:

```
for batch_features, batch_labels in train_loader:
    print(f"Batch Features: {batch_features.shape}")
    print(f"Batch Labels: {batch_labels.shape}")
    break  # Nur ersten Batch zeigen
```

CrossEntropyLoss für mehrere Klassen

Für Multi-Klassen-Klassifikation verwenden wir `CrossEntropyLoss`. Diese Loss-Funktion misst, wie weit die vorhergesagte Wahrscheinlichkeitsverteilung von der tatsächlichen Klasse entfernt ist.

Was bedeutet das konkret? Angenommen, das Netzwerk gibt für ein Weinbeispiel die Werte [2.5, 1.0, 0.3] aus (einen Wert pro Klasse). Diese Rohwerte nennt man **Logits** – sie sind noch keine Wahrscheinlichkeiten, weil sie sich nicht zu 1 summieren und negativ sein können. Softmax transformiert diese Logits in echte Wahrscheinlichkeiten: [0.78, 0.17, 0.05]. Das Netzwerk ist also zu 78% sicher, dass es Klasse 0 ist.

Der **Log-Likelihood** misst dann, wie gut diese Vorhersage ist: Wenn die echte Klasse 0 ist, nehmen wir -log(0.78) ≈ 0.25. Je höher die vorhergesagte Wahrscheinlichkeit für die richtige Klasse, desto kleiner der Loss. Wenn das Netzwerk falsch liegt und nur 5% Wahrscheinlichkeit für die richtige Klasse vorhersagt, ist der Loss -log(0.05) ≈ 3.0, also deutlich höher.

`CrossEntropyLoss` in PyTorch kombiniert Softmax und Log-Likelihood in einer numerisch stabilen Implementierung. Deshalb: - Das Netzwerk gibt **rohe Logits** aus (keine Softmax-Aktivierung in der Ausgabeschicht nötig, das übernimmt `CrossEntropyLoss`) - Die **Targets** (also die erwarteten Ausgaben, die "richtigen Antworten" aus unseren Trainingsdaten) werden als **Klassen-Indizes** angegeben, nicht als One-Hot-Vektoren

Was bedeutet das? Es gibt zwei Möglichkeiten, eine Klasse zu repräsentieren. Bei drei Weinsorten könnte "Sorte 1" entweder als einfacher **Index** `1` dargestellt werden, oder als **One-Hot-Vektor** `[0, 1, 0]` – ein Vektor mit einer 1 an der Position der Klasse und Nullen sonst. Viele andere Frameworks erwarten One-Hot-Vektoren, aber PyTorchs `CrossEntropyLoss` arbeitet direkt mit Indizes. Das ist speichereffizienter und einfacher zu handhaben: Statt `targets = [[1,0,0], [0,1,0], [0,0,1]]` schreiben wir einfach `targets = [0, 1, 2]`.

```
criterion = nn.CrossEntropyLoss()
```

Softmax-Aktivierung

Softmax transformiert einen Vektor von beliebigen Zahlen in eine Wahrscheinlichkeitsverteilung:

```
softmax(x_i) = e^(x_i) / Summe(e^(x_j))
```

Die Ausgaben summieren sich zu 1 und können als Wahrscheinlichkeiten interpretiert werden. Beispiel: [2.0, 1.0, 0.1] wird zu [0.659, 0.242, 0.099] – das Modell ist zu 66% sicher, dass es Klasse 0 ist.

Da `CrossEntropyLoss` Softmax bereits intern anwendet, fügen wir es nicht in unser Modell ein.

Der vollständige Klassifikator

Hier ist das komplette Modell für die Wine-Klassifikation:

```
class WineClassifier(nn.Module):
    def __init__(self, input_size, hidden_sizes, num_classes):
        super().__init__()

        layers = []
        prev_size = input_size
```

```
        for hidden_size in hidden_sizes:
            layers.append(nn.Linear(prev_size, hidden_size))
            layers.append(nn.ReLU())
            prev_size = hidden_size

        # Ausgabeschicht (keine Aktivierung, CrossEntropyLoss übernimmt das)
        layers.append(nn.Linear(prev_size, num_classes))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)


# Modell erstellen
input_size = X_train_tensor.shape[1]  # 13 Features
hidden_sizes = [64, 32]               # Zwei versteckte Schichten
num_classes = 3                       # Drei Weinsorten

model = WineClassifier(input_size, hidden_sizes, num_classes)
print(model)
```

Dropout zur Regularisierung

Wenn ein Modell die Trainingsdaten zu gut lernt, kann es auf neuen Daten schlecht abschneiden. Das nennt man *Overfitting*. Dropout ist eine einfache Technik zur Bekämpfung von Overfitting.

Während des Trainings werden zufällig ausgewählte Neuronen “abgeschaltet” (ihre Ausgabe wird auf 0 gesetzt). Typischerweise werden 20-50% der Neuronen deaktiviert. Das zwingt das Netzwerk, robuste Features zu lernen, die nicht von einzelnen Neuronen abhängen. Historisch war es eine überraschende Erkenntis, dass das “abschalten” bestimmer Parameter beim Training hilft!

```
class WineClassifierWithDropout(nn.Module):
    def __init__(self, input_size, hidden_sizes, num_classes, dropout_rate=0.2):
        super().__init__()

        layers = []
        prev_size = input_size

        for hidden_size in hidden_sizes:
            layers.append(nn.Linear(prev_size, hidden_size))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout_rate))  # Dropout hinzufügen
            prev_size = hidden_size

        layers.append(nn.Linear(prev_size, num_classes))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
```

```
        return self.network(x)
```

Wichtig: Dropout ist nur während des Trainings aktiv. In `model.eval()` wird es automatisch deaktiviert.

Die vollständige Trainingsschleife

Hier ist das komplette Training für den Wine-Klassifikator:

```
import torch.optim as optim


def train_classifier(model, train_loader, val_loader, epochs=100):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    train_losses = []
    val_accuracies = []

    for epoch in range(epochs):
        # === Trainingsphase ===
        model.train()
        total_train_loss = 0
        correct_train = 0
        total_train = 0

        for batch_x, batch_y in train_loader:
            # Forward Pass
            outputs = model(batch_x)
            loss = criterion(outputs, batch_y)

            # Backward Pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Statistiken sammeln
            total_train_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total_train += batch_y.size(0)
            correct_train += (predicted == batch_y).sum().item()

        avg_train_loss = total_train_loss / len(train_loader)
        train_accuracy = correct_train / total_train

        # === Validierungsphase ===
        model.eval()
        correct_val = 0
        total_val = 0

        with torch.no_grad():
            for batch_x, batch_y in val_loader:
```

```
                outputs = model(batch_x)
                _, predicted = torch.max(outputs.data, 1)
                total_val += batch_y.size(0)
                correct_val += (predicted == batch_y).sum().item()

        val_accuracy = correct_val / total_val

        train_losses.append(avg_train_loss)
        val_accuracies.append(val_accuracy)

        if epoch % 10 == 0:
            print(f"Epoch {epoch}: Loss={avg_train_loss:.4f}, "
                  f"Train Acc={train_accuracy:.4f}, Val Acc={val_accuracy:.4f}")

    return train_losses, val_accuracies


# Training starten
model = WineClassifierWithDropout(input_size, hidden_sizes, num_classes)
train_losses, val_accs = train_classifier(model, train_loader, val_loader)
```

Der Code verwendet `Adam` statt SGD. Adam (Adaptive Moment Estimation) passt die Lernrate für jeden Parameter individuell an und konvergiert oft schneller als SGD. Adam basiert auf zwei Ideen: Erstens speichert es einen gleitenden Durchschnitt der Gradienten (Momentum), um Schwankungen auszugleichen. Wenn die Gradienten konsistent in eine Richtung zeigen, bewegt sich Adam schneller in diese Richtung. Zweitens speichert es einen gleitenden Durchschnitt der quadrierten Gradienten, um die Lernrate anzupassen: Parameter, deren Gradienten stark schwanken, bekommen eine kleinere effektive Lernrate, während Parameter mit konsistenten, kleinen Gradienten schneller lernen. Diese Kombination macht Adam zu einer robusten Standardwahl, die bei vielen Problemen ohne aufwendiges Tuning der Lernrate funktioniert.

Die Zeile `_, predicted = torch.max(outputs.data, 1)` verdient eine Erklärung: `torch.max()` gibt sowohl den Maximalwert als auch seinen Index zurück. Da uns nur der Index interessiert (welche Klasse hat die höchste Wahrscheinlichkeit?), ignorieren wir den Wert mit `_`.

Nach dem Training sollte die Validierungsgenauigkeit bei über 95% liegen – unser Netzwerk kann verschiedene Weinsorten anhand ihrer chemischen Zusammensetzung unterscheiden!

Zusammenfassung

In diesem Kapitel haben Sie die Grundlagen neuronaler Netze kennengelernt:

- **Das Perzeptron** ist die einfachste Form eines neuronalen Netzes, kann aber nur linear trennbare Probleme lösen.
- **Multi-Layer Perceptrons** überwinden diese Einschränkung durch versteckte Schichten und nicht-lineare Aktivierungsfunktionen.
- **Backpropagation** ermöglicht effizientes Training durch automatische Gradientenberechnung.

- **Das torch.nn Modul** bietet Bausteine wie `nn.Linear` und `nn.Module` für sauberen, wartbaren Code.
- **DataLoader** und **Normalisierung** sind essentiell für die Arbeit mit echten Datensätzen.
- **CrossEntropyLoss** und **Softmax** werden für Multi-Klassen-Klassifikation verwendet.
- **Dropout** hilft, Overfitting zu vermeiden.

Die Konzepte und Muster, die Sie hier gelernt haben, bilden die Grundlage für alle weiteren Architekturen. Im nächsten Kapitel werden wir spezialisiertere Netzwerktypen kennenlernen: Convolutional Neural Networks (CNNs) für Bilder und Recurrent Neural Networks (RNNs) für sequentielle Daten.

Kapitel 3: Convolutional und Recurrent Neural Networks

In den ersten beiden Kapiteln haben Sie die Grundlagen von PyTorch und neuronalen Netzen kennengelernt. Mit Multi-Layer Perceptrons können Sie bereits viele Probleme lösen, aber manche Datentypen erfordern spezialisierte Architekturen. In diesem Kapitel lernen Sie zwei wichtige Netzwerkfamilien kennen: Convolutional Neural Networks (CNNs) für Bilder und Recurrent Neural Networks (RNNs) für sequentielle Daten wie Text.

3.1 CNNs: Grundlagen der Bildverarbeitung

CNNs für Computer Vision

Stellen Sie sich vor, Sie wollen ein neuronales Netz trainieren, das Bilder von Katzen und Hunden unterscheidet. Mit einem herkömmlichen Multi-Layer Perceptron müssten Sie das Bild zunächst "flach machen" – aus einem 32×32-Pixel-Bild mit 3 Farbkanälen wird ein Vektor mit `32*32*3=3072` Werten. Dabei geht die räumliche Struktur verloren: Das Netzwerk weiß nicht mehr, welche Pixel nebeneinander liegen.

Convolutional Neural Networks (CNNs) lösen dieses Problem elegant. Sie sind speziell für gitterförmige Daten wie Bilder entwickelt worden und nutzen drei zentrale Ideen:

1. **Lokale Verbindungen**: Jedes Neuron betrachtet nur einen kleinen Ausschnitt des Bildes (sein "rezeptives Feld")
2. **Geteilte Gewichte**: Dieselben Gewichte werden über das gesamte Bild verwendet, um Muster unabhängig von ihrer Position zu erkennen
3. **Hierarchische Feature-Extraktion**: Frühe Schichten erkennen einfache Muster (Kanten, Linien), spätere Schichten komplexere Strukturen (Augen, Ohren, Gesichter)

Die Anwendungsgebiete von CNNs sind vielfältig: Bildklassifikation (ist das eine Katze oder ein Hund?), Objekterkennung (wo im Bild ist die Katze?), medizinische Bildanalyse (Tumoren in Röntgenbildern erkennen), autonomes Fahren (Fußgänger und Verkehrsschilder erkennen) und vieles mehr.

Convolution-Operationen verstehen

Das Herzstück eines CNNs ist die Convolution-Operation (Faltung). Die Idee ist einfach: Ein kleines Gewichtsmuster, der sogenannte *Filter* oder *Kernel*, gleitet

über das Bild. An jeder Position wird der Filter elementweise mit dem darunterliegenden Bildausschnitt multipliziert und die Ergebnisse werden summiert.

Ein 3×3-Filter für Kantenerkennung könnte so aussehen:

```
[-1, -1, -1]
[-1,  8, -1]
[-1, -1, -1]
```

Wenn dieser Filter über einen einheitlich gefärbten Bereich gleitet, ist das Ergebnis nahe Null. An einer Kante, wo sich die Pixelwerte abrupt ändern, wird das Ergebnis groß. Der Filter "reagiert" also auf Kanten.

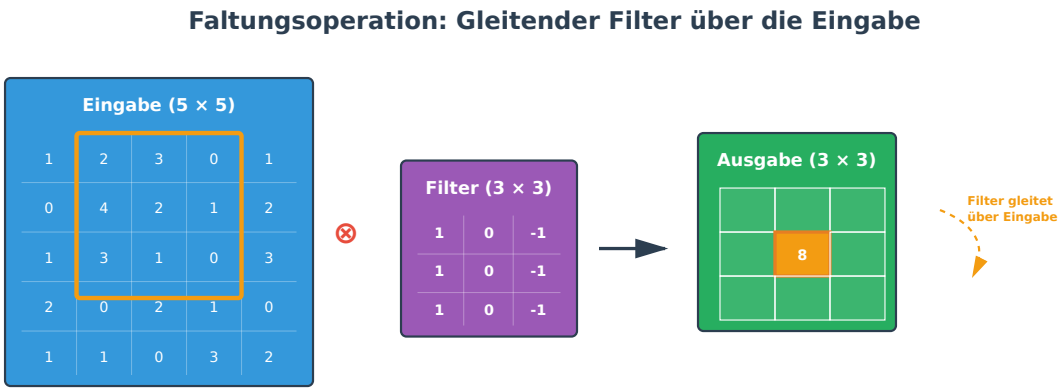

Abbildung 4: Die Convolution-Operation: Ein 3×3-Filter gleitet über das Eingabebild

Die Convolution hat zwei wichtige Parameter:

Stride bestimmt, wie weit der Filter bei jedem Schritt bewegt wird. Ein Stride von 1 bedeutet, der Filter gleitet Pixel für Pixel. Ein Stride von 2 überspringt jeden zweiten Pixel und halbiert damit die Ausgabegröße.

Padding fügt zusätzliche Werte (meist Nullen) am Rand des Bildes hinzu. Ohne Padding wird die Ausgabe bei jedem Convolution-Schritt kleiner, da wir den Filter nicht über den Rand hinaus schieben können. Mit "same padding" bleibt die räumliche Größe erhalten.

Hier ist ein einfaches Beispiel in PyTorch:

```
import torch
import torch.nn as nn

torch.manual_seed(42)

# Convolutional Layer definieren
conv_layer = nn.Conv2d(
```

```
    in_channels=3,   # RGB-Eingabe (3 Farbkanäle)
    out_channels=16, # 16 verschiedene Filter (Feature Maps)
    kernel_size=3,   # 3x3 Filter
    stride=1,        # Ein Pixel pro Schritt
    padding=1,       # Padding für gleiche Ausgabegröße
)

# Beispielbild erstellen (Batch=1, Kanäle=3, Höhe=32, Breite=32)
input_image = torch.randn(1, 3, 32, 32)

# Convolution anwenden
feature_maps = conv_layer(input_image)

print(f"Eingabe-Shape: {input_image.shape}")
print(f"Ausgabe-Shape: {feature_maps.shape}")
```

Die Ausgabe zeigt:

```
Eingabe-Shape: torch.Size([1, 3, 32, 32])
Ausgabe-Shape: torch.Size([1, 16, 32, 32])
```

Das Bild hat nach der Convolution dieselbe räumliche Größe (32×32), aber statt 3 Farbkanälen haben wir jetzt 16 Feature Maps. Diese Erweiterung der Kanalzahl ist ein häufig verwendetes Muster in CNNs. Die Motivation dahinter: Während die RGB-Farbkanäle nur drei verschiedene Aspekte des Bildes darstellen (Rot, Grün, Blau), können 16, 32 oder sogar Hunderte von Feature Maps viele verschiedene Muster gleichzeitig kodieren: horizontale Kanten, vertikale Kanten, Ecken, Texturen und mehr.

Die Wahl der Kanalzahl folgt typischerweise einer Faustregel: Man beginnt mit einer moderaten Anzahl (z.B. 32 oder 64) und verdoppelt die Kanäle nach jedem Schritt, während die räumliche Auflösung halbiert wird (s.u. Pooling). Die genaue Anzahl hängt von der Komplexität der Aufgabe ab. Einfache Aufgaben wie die Zifferneerkennung benötigen weniger Kanäle als komplexe Aufgaben wie die Objekterkennung in natürlichen Szenen. Jede Feature Map ist das Ergebnis eines anderen Filters und repräsentiert ein bestimmtes Muster im Bild.

Die Dimensionen eines Tensors in CNNs folgen der Konvention `[Batch, Kanäle, Höhe, Breite]`. Der Batch ist die Anzahl der Bilder, die gleichzeitig verarbeitet werden. Die Kanäle sind bei der Eingabe typischerweise die Farbkanäle (3 für RGB), bei späteren Schichten die Feature Maps.

Pooling: Max Pool vs. Average Pool

Nach einer Convolution-Schicht wird oft eine Pooling-Schicht eingefügt. Pooling reduziert die räumliche Größe der Feature Maps und macht die Repräsentation robuster gegenüber kleinen Verschiebungen.

Max Pooling nimmt den maximalen Wert aus jedem Bereich. Ein 2×2 Max Pooling mit Stride 2 unterteilt die Feature Map in 2×2-Blöcke und behält jeweils nur den größten Wert. Die Ausgabe ist halb so groß wie die Eingabe.

Average Pooling berechnet stattdessen den Durchschnitt jedes Bereichs. Es ist “sanfter” als Max Pooling, verliert aber möglicherweise markante Merkmale.

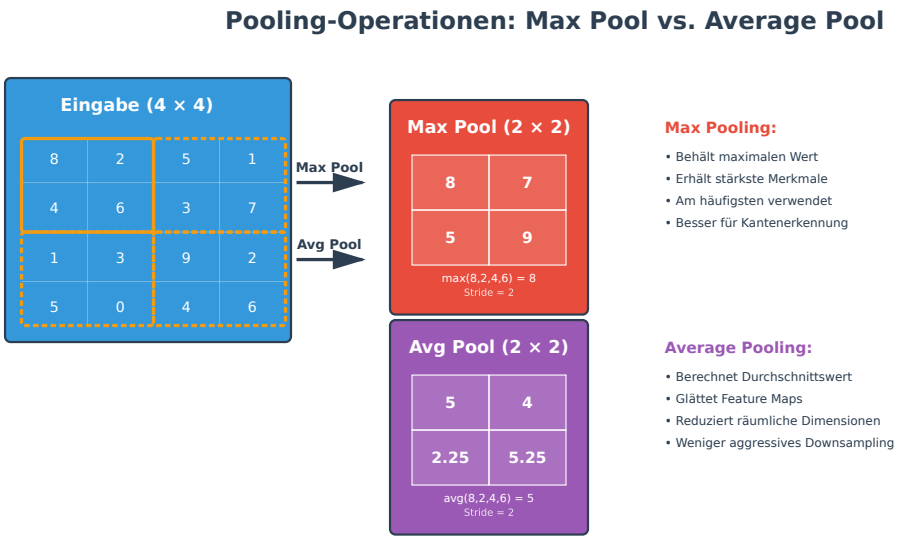

Abbildung 5: Max Pooling vs. Average Pooling: Beide reduzieren die räumliche Größe

Max Pooling ist in der Praxis häufiger, weil es die stärksten Aktivierungen (die auffälligsten Merkmale) erhält:

```
# Pooling-Layer definieren
max_pool = nn.MaxPool2d(kernel_size=2, stride=2)
avg_pool = nn.AvgPool2d(kernel_size=2, stride=2)

# Pooling anwenden
pooled_max = max_pool(feature_maps)
pooled_avg = avg_pool(feature_maps)

print(f"Original Feature Maps: {feature_maps.shape}")
print(f"Nach Max Pooling: {pooled_max.shape}")
print(f"Nach Average Pooling: {pooled_avg.shape}")
```

```
Original Feature Maps: torch.Size([1, 16, 32, 32])
Nach Max Pooling: torch.Size([1, 16, 16, 16])
Nach Average Pooling: torch.Size([1, 16, 16, 16])
```

Beide Pooling-Varianten halbieren die räumliche Größe von 32×32 auf 16×16. Die Anzahl der Kanäle bleibt unverändert.

CNN-Architektur-Design

Ein typisches CNN folgt einem wiederkehrenden Muster:

`Conv2d → BatchNorm2d → ReLU → MaxPool2d → (wiederholen) → Flatten → Linear`

Die Idee: Mit jeder Convolution+Pooling-Stufe wird die räumliche Auflösung kleiner, aber die Anzahl der Feature Maps (Kanäle) nimmt zu. Das Netzwerk komprimiert das Bild schrittweise von einer hochauflösenden, aber “flachen” Darstellung (wenige Kanäle) zu einer niedrig auflösenden, aber “tiefen” Darstellung (viele Kanäle).

Batch Normalization (`nn.BatchNorm2d`) normalisiert die Aktivierungen innerhalb

eines Batches. Zur Erinnerung aus Kapitel 2: Normalisierung bedeutet, dass die Werte so transformiert werden, dass sie einen Mittelwert nahe 0 und eine Standardabweichung nahe 1 haben. Das stabilisiert das Training und ermöglicht höhere Lernraten. Die Idee: Wenn die Eingaben einer Schicht stark schwanken, muss die nächste Schicht ständig neu "kalibrieren". Batch Normalization hält die Verteilung der Aktivierungen stabil und beschleunigt dadurch die Konvergenz des Trainings erheblich.

Flatten bezeichnet den Schritt, bei dem die mehrdimensionalen Feature Maps zu einem eindimensionalen Vektor zusammengeführt werden. Nach mehreren Convolution- und Pooling-Schichten haben wir einen Tensor der Form `[Batch, Kanäle, Höhe, Breite]`. Für die finale Klassifikation benötigen wir jedoch einen flachen Vektor, also `[Batch, Kanäle × Höhe × Breite]`. Alle Parameter bleiben dabei erhalten, diese werden in einen eindimensionalen Vektor "kopiert" (ohne dass dabei wirklich Werte im Speicher kopiert werden, wegen Effizienz).

Linear (auch "Fully Connected" oder "Dense" genannt) ist die klassische Schicht aus dem Multi-Layer Perceptron: Jeder Eingabewert ist mit jedem Ausgabewert verbunden. Nach dem Flatten übernimmt die Linear-Schicht die eigentliche Klassifikation, indem sie die extrahierten Features auf die Ausgabeklassen abbildet.

Hier ist eine einfache CNN-Klasse:

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()

        # Erster Convolution-Block
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Klassifikator
        self.dropout = nn.Dropout(0.5)
        self.classify = nn.Linear(32 * 16 * 16, num_classes)

    def forward(self, x):
        # Convolution-Block
        x = torch.relu(self.bn1(self.conv1(x)))
        x = self.pool1(x)

        # Flach machen und klassifizieren
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = self.classify(x)

        return x
```

Die Zeile `x = x.view(x.size(0), -1)` verdient Beachtung: Sie formt den Tensor von `[Batch, Kanäle, Höhe, Breite]` zu `[Batch, Kanäle*Höhe*Breite]` um. Das ist nötig, weil die finale `nn.Linear`-Schicht einen flachen Vektor erwartet. Das `-1` lässt PyTorch die letzte Dimension automatisch berechnen.

3.2 CNNs in der Praxis

CIFAR-10 Dataset

Um CNNs praktisch zu erlernen, verwenden wir den CIFAR-10-Datensatz. Er ist ein Standardbenchmark für Bildklassifikation mit überschaubarer Komplexität:

- 60.000 Farbbilder in 10 Klassen
- Jedes Bild ist 32×32 Pixel groß mit 3 RGB-Kanälen
- 50.000 Trainingsbilder und 10.000 Testbilder
- Die Klassen: Flugzeug, Auto, Vogel, Katze, Hirsch, Hund, Frosch, Pferd, Schiff, LKW

PyTorch bietet über `torchvision` einfachen Zugriff auf CIFAR-10. Wir brauchen die folgenden Imports:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

torch.manual_seed(42)
```

Data Augmentation und Normalisierung

Eine wichtige Technik beim Training von CNNs ist *Data Augmentation* (Datenaugmentierung). Die Idee: Wir erzeugen künstlich mehr Trainingsdaten, indem wir die vorhandenen Bilder leicht verändern: spiegeln, drehen, zuschneiden, die Helligkeit anpassen. Das Netzwerk lernt dadurch, robuster gegenüber Variationen zu sein.

```
# Transformationen für Trainingsdaten (mit Augmentation)
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),  # Zufällig horizontal spiegeln
    transforms.RandomCrop(32, padding=4),  # Zufällig zuschneiden
    transforms.ToTensor(),
    transforms.Normalize(
        (0.4914, 0.4822, 0.4465),  # Mittelwerte pro Kanal
        (0.2470, 0.2435, 0.2616)   # Standardabweichungen pro Kanal
    ),
])

# Transformationen für Validierungsdaten (ohne Augmentation)
transform_validation = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(
        (0.4914, 0.4822, 0.4465),
        (0.2470, 0.2435, 0.2616)
    ),
])
```

Die Normalisierungswerte sind die Mittelwerte und Standardabweichungen der RGB-Kanäle über den gesamten CIFAR-10-Datensatz. Diese Werte sind bekannt

und werden standardmäßig verwendet.

Data Augmentation wird nur auf die Trainingsdaten angewendet. Die Validierungsdaten bleiben unverändert, damit wir die tatsächliche Leistung des Modells messen können. Wir laden zunächst das Dataset und erstellen damit einen DataLoader, so wie wir das in Kapitel 2 besprochen haben.

```
# Datensätze laden
print("Lade CIFAR-10 Datensatz...")
train_dataset = torchvision.datasets.CIFAR10(
    root="./data", train=True, download=True, transform=transform_train
)
validation_dataset = torchvision.datasets.CIFAR10(
    root="./data", train=False, download=True, transform=transform_validation
)

# DataLoader erstellen
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
validation_loader = DataLoader(validation_dataset, batch_size=100, shuffle=False)

# Klassennamen
classes = ("Flugzeug", "Auto", "Vogel", "Katze", "Hirsch",
           "Hund", "Frosch", "Pferd", "Schiff", "LKW")

print(f"Trainingsbeispiele: {len(train_dataset)}")
print(f"Validierungsbeispiele: {len(validation_dataset)}")
```

Ein vollständiges CNN für CIFAR-10

Hier ist ein CNN mit drei Convolution-Blöcken, das für CIFAR-10 gut funktioniert:

```
class CIFAR10CNN(nn.Module):
    def __init__(self, num_classes=10):
        super(CIFAR10CNN, self).__init__()

        # Erster Convolution-Block: 3 → 32 Kanäle
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Zweiter Convolution-Block: 32 → 64 Kanäle
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Dritter Convolution-Block: 64 → 128 Kanäle
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Klassifikator
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(128 * 4 * 4, num_classes)
```

```
    def forward(self, x):
        # Block 1: 32x32 → 16x16
        x = torch.relu(self.bn1(self.conv1(x)))
        x = self.pool1(x)

        # Block 2: 16x16 → 8x8
        x = torch.relu(self.bn2(self.conv2(x)))
        x = self.pool2(x)

        # Block 3: 8x8 → 4x4
        x = torch.relu(self.bn3(self.conv3(x)))
        x = self.pool3(x)

        # Flach machen und klassifizieren
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = self.fc(x)

        return x
```

Beachten Sie, wie die räumliche Größe abnimmt (32→16→8→4) während die Kanalzahl zunimmt (3→32→64→128). Am Ende haben wir 128 Feature Maps mit je 4×4 Pixeln, also 128×4×4 = 2048 Werte pro Bild, die in die finale Linear-Schicht gehen.

```
# Modell erstellen und Struktur anzeigen
model = CIFAR10CNN(num_classes=10)
test_input = torch.randn(4, 3, 32, 32)  # Batch mit 4 Bildern
output = model(test_input)

print(f"Ausgabe-Shape: {output.shape}")
print(f"Anzahl Parameter: {sum(p.numel() for p in model.parameters()):,}")
```

Training und Evaluation

Die Trainingsschleife für CNNs folgt demselben Muster wie bei MLPs:

```
def train_epoch(model, dataloader, loss_fn, optimizer):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for batch_idx, (data, targets) in enumerate(dataloader):
        optimizer.zero_grad()

        # Forward Pass
        outputs = model(data)
        loss = loss_fn(outputs, targets)

        # Backward Pass
        loss.backward()
```

```
        optimizer.step()

        # Statistiken sammeln
        running_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()

        if batch_idx % 100 == 0:
            print(f"Batch {batch_idx}, Loss: {loss.item():.4f}, "
                  f"Acc: {100.0 * correct / total:.2f}%")

    return running_loss / len(dataloader), 100.0 * correct / total


def validate_epoch(model, dataloader, loss_fn):
    model.eval()
    test_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for data, targets in dataloader:
            outputs = model(data)
            loss = loss_fn(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    return test_loss / len(dataloader), 100.0 * correct / total
```

Nun können wir das Training starten:

```
def train(model, train_loader, validation_loader, epochs=5):
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    for epoch in range(epochs):
        print(f"\nEpoch {epoch + 1}/{epochs}")

        # Training
        train_loss, train_acc = train_epoch(model, train_loader, loss_fn, optimizer)

        # Validierung
        val_loss, val_acc = validate_epoch(model, validation_loader, loss_fn)

        print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%")
        print(f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%")
```

```
# Training starten
train(model, train_loader, validation_loader, epochs=5)
```

Nach wenigen Epochen sollten Sie eine Validierungsgenauigkeit von etwa 70-75% erreichen. Mit längerer Trainingszeit, mehr Schichten und fortgeschrittenen Techniken wie Learning Rate Scheduling sind über 90% möglich.

3.3 RNN-Grundlagen und Embeddings

Sequenzielle Daten verstehen

CNNs sind großartig für Bilder, aber was ist mit Text? Text ist *sequenziell*: Die Reihenfolge der Wörter ist entscheidend. "Hund beißt Mann" bedeutet etwas völlig anderes als "Mann beißt Hund".

Sequenzielle Daten haben besondere Eigenschaften: - **Reihenfolge ist wichtig**: Die Position eines Elements beeinflusst seine Bedeutung - **Variable Länge**: Sätze können 5 oder 500 Wörter lang sein - **Abhängigkeiten**: Das aktuelle Wort hängt oft von vorherigen Wörtern ab

Beispiele für sequenzielle Daten sind Text, Sprache, Zeitreihen (Aktienkurse, Wetterdaten), DNA-Sequenzen und Musiknoten.

Multi-Layer Perceptrons können sequenzielle Daten nicht gut verarbeiten, weil sie eine feste Eingabegröße erwarten und keine "Erinnerung" an vorherige Eingaben haben. Recurrent Neural Networks (RNNs) wurden speziell für dieses Problem entwickelt.

Word Embeddings: Von Wörtern zu Vektoren

Neuronale Netze arbeiten mit Zahlen, nicht mit Wörtern. Wie wandeln wir Text in Zahlen um?

Die einfachste Methode wäre *One-Hot-Encoding*: Jedes Wort wird als Vektor dargestellt, der überall Null ist außer an einer Stelle. Bei einem Vokabular von 10.000 Wörtern wäre das ein 10.000-dimensionaler Vektor mit einer einzigen 1. Das ist extrem ineffizient und enthält keine Information über Wortbeziehungen – "König" und "Königin" wären genauso "verschieden" wie "König" und "Gurke".

Word Embeddings lösen dieses Problem elegant. Jedes Wort wird auf einen dichten Vektor mit typischerweise 100-300 Dimensionen abgebildet. Diese Vektoren werden gelernt, sodass ähnliche Wörter ähnliche Vektoren haben:

- "König" und "Königin" haben ähnliche Vektoren
- "laufen" und "rennen" haben ähnliche Vektoren
- Man kann sogar rechnen: König - Mann + Frau ≈ Königin

PyTorch bietet `nn.Embedding` für diesen Zweck:

```
import torch
import torch.nn as nn

torch.manual_seed(42)

# Embedding-Layer: 10.000 Wörter, je 128 Dimensionen
embedding = nn.Embedding(num_embeddings=10000, embedding_dim=128)
```

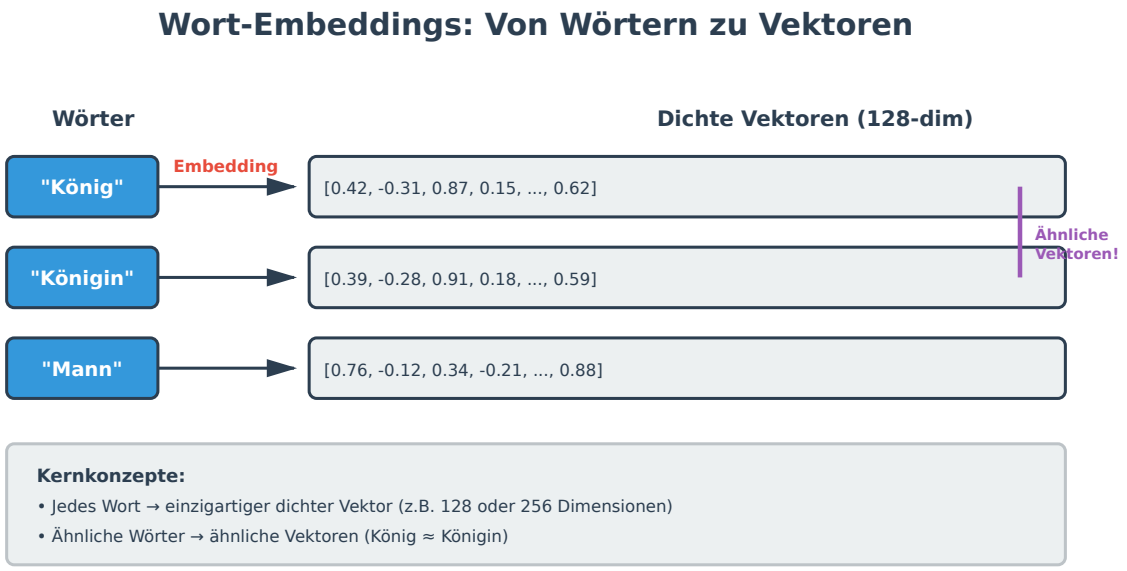

Abbildung 6: Word Embeddings: Ähnliche Wörter liegen im Vektorraum nahe beieinander

```
# Beispiel: Zwei Sätze mit je 5 Wort-Indizes
word_indices = torch.tensor([
    [1, 45, 2, 99, 3],
    [8, 12, 2, 1, 77]
])

# Embeddings abrufen
embedded = embedding(word_indices)

print(f"Eingabe-Shape: {word_indices.shape}")
print(f"Ausgabe-Shape: {embedded.shape}")
```

```
Eingabe-Shape: torch.Size([2, 5])
Ausgabe-Shape: torch.Size([2, 5, 128])
```

Aus einem Tensor mit Wort-Indizes `[Batch, Sequenzlänge]` wird ein Tensor mit Embedding-Vektoren `[Batch, Sequenzlänge, Embedding-Dimension]`. Jeder Index wird durch seinen entsprechenden 128-dimensionalen Vektor ersetzt. Dazu wird der Index eines Wortes im Vokublar verwendet, dass wir vorher anlegen. Meist beschränkt man das Vokabular auf die häufigsten 10.000 oder 100.000 Wörter einer Sprache. Dazu zählt man die Wörter in einem sogenannten Korpus, einer Sammlung aus Dokumenten wie z.B. Wikipedia oder eine Dokumenten-Datenbank in einer Firma.

Die Embedding-Schicht ist nichts anderes als eine trainierbare Lookup-Tabelle: Für jeden Index gibt es einen Vektor, der während des Trainings angepasst wird. Alternativ können Sie vortrainierte Embeddings wie Word2Vec oder GloVe verwenden, die auf großen Textkorpora gelernt wurden.

Die RNN-Zelle: Kernkonzept

Ein RNN verarbeitet Sequenzen Element für Element. Das Besondere: Es hat einen *Hidden State* (verborgenen Zustand), der Informationen von einem Schritt zum nächsten weitergibt. Sie können sich den Hidden State als "Erinnerung" des Netzwerks vorstellen.

Bei jedem Zeitschritt: 1. Das RNN erhält die aktuelle Eingabe und den vorherigen Hidden State 2. Es berechnet einen neuen Hidden State 3. Der neue Hidden State wird für den nächsten Schritt gespeichert

Die Formel für eine einfache RNN-Zelle ist:

`h_t = tanh(W_h · h_{t-1} + W_x · x_t + b)`

Dabei ist: - `h_t`: Der neue Hidden State zum Zeitpunkt t - `h_{t-1}`: Der vorherige Hidden State - `x_t`: Die aktuelle Eingabe - `W_h`, `W_x`: Gewichtsmatrizen - `b`: Bias - `tanh`: Die Aktivierungsfunktion (hält Werte zwischen -1 und 1)

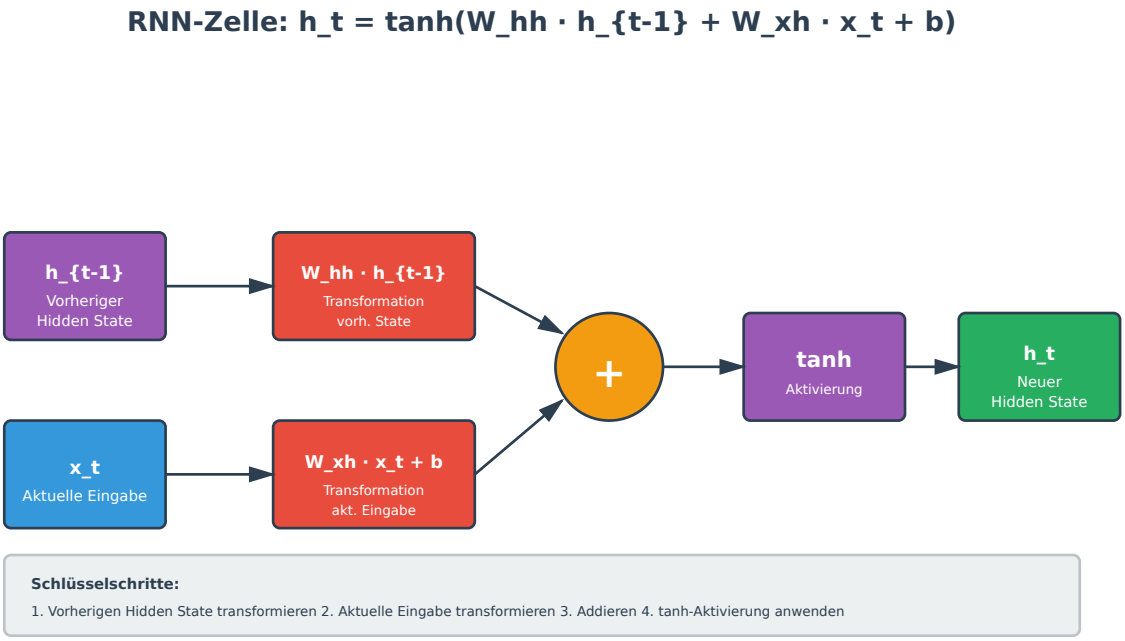

Abbildung 7: Die RNN-Zelle: Der Hidden State wird von Zeitschritt zu Zeitschritt weitergegeben

RNN vs. Linear Layers

Eine RNN-Zelle ist eigentlich nur die Kombination von zwei linearen Transformationen mit einer tanh-Aktivierung. Der Unterschied zu einem normalen MLP: Eine der Eingaben ist der *eigene vorherige Zustand*. Das macht das Netzwerk rekurrent (zurückkehrend).

Hier ist eine manuelle Implementierung einer RNN-Zelle:

```
class ManualRNNCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.W_h = nn.Linear(hidden_size, hidden_size, bias=True)
```

```
        self.W_x = nn.Linear(input_size, hidden_size, bias=True)

    def forward(self, x_t, h_prev):
        h_new = torch.tanh(self.W_h(h_prev) + self.W_x(x_t))
        return h_new
```

Das ist alles! Die Magie des RNNs liegt nicht in komplizierter Mathematik, sondern darin, dass derselbe Zustand Schritt für Schritt weitergegeben wird.

PyTorch bietet natürlich eine fertige Implementierung:

```
# PyTorchs RNNCell
rnn_cell = nn.RNNCell(input_size=4, hidden_size=8)

# Beispiel: Ein Eingabevektor und ein initialer Hidden State
x_t = torch.randn(2, 4)   # Batch=2, Features=4
h_t = torch.zeros(2, 8)   # Batch=2, Hidden=8

# Einen Schritt ausführen
h_new = rnn_cell(x_t, h_t)
print(f"Neuer Hidden State: {h_new.shape}")
```

Im Beispiel oben hat der Hidden State 8 Dimensionen. In der Praxis verwendet man oft deutlich mehr, typischerweise 128, 256 oder sogar 512 Dimensionen. Die Motivation: Der Hidden State muss alle relevanten Informationen der bisherigen Sequenz speichern. Je komplexer die Aufgabe, desto mehr "Speicherplatz" benötigt das Netzwerk. Für einfache Aufgaben wie das Zählen von Zeichen reichen wenige Dimensionen; für das Verstehen komplexer Sätze braucht man mehr. Die Wahl der Hidden-Size ist ein Kompromiss zwischen Ausdrucksstärke und Rechenaufwand. Mehr Dimensionen bedeuten mehr Parameter und längere Trainingszeiten.

Eine vollständige Sequenz verarbeiten

Um eine ganze Sequenz zu verarbeiten, wenden wir die RNN-Zelle in einer Schleife an:

```
def process_sequence(rnn_cell, sequence, initial_hidden):
    batch_size, seq_length, input_size = sequence.shape
    hidden = initial_hidden
    outputs = []

    for t in range(seq_length):
        x_t = sequence[:, t, :]  # Eingabe zum Zeitpunkt t
        hidden = rnn_cell(x_t, hidden)
        outputs.append(hidden)

    # Alle Hidden States zu einem Tensor zusammenfügen
    all_outputs = torch.stack(outputs, dim=1)
    return all_outputs, hidden


# Beispiel: Zufällige Sequenz mit 10 Wörtern und einem Embedding mit Dimension 4
sequence = torch.randn(32, 10, 4)  # Batch=32, Länge=10, Features=4
h0 = torch.zeros(32, 8)            # Initialer Hidden State
```

```
all_outputs, final_hidden = process_sequence(rnn_cell, sequence, h0)

print(f"Sequenz: {sequence.shape}")
print(f"Alle Outputs: {all_outputs.shape}")
print(f"Finaler Hidden State: {final_hidden.shape}")
```

```
Sequenz: torch.Size([32, 10, 4])
Alle Outputs: torch.Size([32, 10, 8])
Finaler Hidden State: torch.Size([32, 8])
```

Die `all_outputs` enthalten den Hidden State nach jedem Zeitschritt. Der `final_hidden` ist der Hidden State nach dem letzten Zeitschritt. Er "fasst" die gesamte Sequenz zusammen und wird oft für Klassifikationsaufgaben verwendet.

PyTorch bietet mit `nn.RNN` eine effiziente Implementierung, die die Schleife intern optimiert:

```
rnn = nn.RNN(input_size=4, hidden_size=8, num_layers=1, batch_first=True)

output, hidden = rnn(sequence, h0.unsqueeze(0))
print(f"Output: {output.shape}")
print(f"Hidden: {hidden.shape}")
```

3.4 Sentiment-Analyse mit gestapelten RNNs

Der IMDB-Datensatz

Für unsere erste praktische RNN-Anwendung verwenden wir den IMDB-Datensatz: 50.000 Filmkritiken, die entweder positiv oder negativ bewertet wurden. Die Aufgabe ist *Sentiment-Analyse*: Aus dem Text einer Kritik vorhersagen, ob sie positiv oder negativ ist.

```
from datasets import load_dataset

dataset = load_dataset("imdb")
print(f"Trainingsbeispiele: {len(dataset['train'])}")
print(f"Testbeispiele: {len(dataset['test'])}")
```

Die Kritiken sind echte Texte mit allen Herausforderungen, die das mit sich bringt: unterschiedliche Längen, Rechtschreibfehler, Slang, Sarkasmus.

Text-Preprocessing und Tokenisierung

Bevor wir Text in ein neuronales Netz geben können, müssen wir ihn aufbereiten:

Tokenisierung zerlegt den Text in Einheiten (Tokens), meist Wörter:

```
import re

def clean_text(text):
    text = text.lower()  # Kleinschreibung
    text = re.sub(r"<br />", " ", text)  # HTML-Tags entfernen
    text = re.sub(r"[^a-zA-Z0-9\s]", "", text)  # Sonderzeichen entfernen
    text = re.sub(r"\s+", " ", text)  # Mehrfache Leerzeichen
```

```
    return text.strip()

def tokenize(text):
    return text.lower().split()

# Beispiel
example = "This movie was AMAZING! Best film I've seen in years..."
cleaned = clean_text(example)
tokens = tokenize(cleaned)
print(f"Original: {example}")
print(f"Bereinigt: {cleaned}")
print(f"Tokens: {tokens}")
```

Die Ausgabe ist dann:

```
Original: This movie was AMAZING! Best film I've seen in years...
Bereinigt: this movie was amazing best film ive seen in years
Tokens: ['this', 'movie', 'was', 'amazing', 'best', 'film', 'ive', 'seen', 'in', 'years'
```

Vokabular aufbauen

Wir brauchen eine Zuordnung von Wörtern zu numerischen Indizes. Das *Vokabular* ist ein Dictionary, das jedem Wort eine eindeutige Zahl zuweist.

Wir fügen dem Vokabular außerdem zwei spezielle Tokens hinzu. Diese haben wichtige Funktionen:

- `<PAD>` (Index 0): Da alle Sequenzen in einem Batch dieselbe Länge haben müssen, werden kürzere Sequenzen mit diesem Token aufgefüllt. Nur dadurch können wir in der Batch-Verarbeitung alle Tensoren auf einmal multiplizieren und addieren. Wenn ein Satz nur 5 Wörter hat, aber die maximale Länge 256 beträgt, werden 251 `<PAD>`-Tokens angehängt. Das Modell lernt, diese Tokens zu ignorieren.
- `<UNK>` (Index 1): Wir wollen das Vokabular, wie oben besprochen, auf eine Maximalzahl begrenzen, damit die Berechnungen nicht zu viele Dimensionen und Parameter benötigen. Dazu steht dieses Token für “Unknown” (unbekannt). Wenn das Modell später auf ein Wort trifft, das nicht im Vokabular ist (z.B. einen seltenen Eigennamen oder einen Tippfehler), wird es durch dieses Token ersetzt. So kann das Modell auch mit unbekannten Wörtern umgehen, ohne einen Fehler zu werfen.

```
from collections import Counter

def build_vocab(texts, max_vocab_size):
    counter = Counter()
    for text in texts:
        counter.update(tokenize(text))

    # Die häufigsten Wörter auswählen
    # Wir ziehen hier 2 ab, wegen den speziellen Tokens
    most_common = counter.most_common(max_vocab_size - 2)

    # Spezielle Tokens
```

```
    vocab = {"<PAD>": 0, "<UNK>": 1}
    for word, _ in most_common:
        vocab[word] = len(vocab)

    return vocab

# Vokabular aus Trainingsdaten erstellen
MAX_VOCAB_SIZE = 10000
train_texts = [clean_text(ex["text"]) for ex in dataset["train"]]
vocab = build_vocab(train_texts, MAX_VOCAB_SIZE)
print(f"Vokabulargröße: {len(vocab)}")
```

Padding und Truncation

RNNs können theoretisch Sequenzen beliebiger Länge verarbeiten, aber in der Praxis müssen alle Sequenzen in einem Batch dieselbe Länge haben. Wir lösen das durch:

Truncation: Zu lange Sequenzen werden auf eine maximale Länge gekürzt **Padding**: Zu kurze Sequenzen werden mit `<PAD>`-Tokens aufgefüllt

Um unsere Daten effizient mit PyTorch verarbeiten zu können, erstellen wir eine eigene Dataset-Klasse, die von `Dataset` erbt. PyTorch erwartet, dass jede Dataset-Klasse zwei Methoden implementiert: `__len__()` gibt die Anzahl der Beispiele zurück, und `__getitem__(idx)` liefert ein einzelnes Beispiel anhand seines Index. Diese Struktur ermöglicht es dem DataLoader, die Daten automatisch zu batchen, zu mischen und parallel zu laden.

```
from torch.utils.data import Dataset, DataLoader

class IMDBDataset(Dataset):
    def __init__(self, texts, labels, vocab, max_length):
        self.texts = texts
        self.labels = labels
        self.vocab = vocab
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        tokens = tokenize(self.texts[idx])

        # Wörter zu Indizes konvertieren
        indices = [self.vocab.get(token, self.vocab["<UNK>"])
                   for token in tokens]

        # Truncation oder Padding
        if len(indices) > self.max_length:
            indices = indices[:self.max_length]
        else:
            indices += [self.vocab["<PAD>"]] * (self.max_length - len(indices))
```

```
        return (
            torch.tensor(indices, dtype=torch.long),
            torch.tensor(self.labels[idx], dtype=torch.long)
        )
```

Gestapelte RNN-Architektur

Ein einzelnes RNN-Layer kann bereits Sequenzen verarbeiten, aber mehrere gestapelte Schichten können komplexere Muster lernen. Die Ausgabe des ersten RNN-Layers wird zur Eingabe des zweiten:

```
Eingabe → RNN Layer 1 → RNN Layer 2 → ... → Ausgabe
```

Jede Schicht lernt unterschiedliche Abstraktionsebenen: Die erste Schicht könnte grundlegende Muster wie "nicht gut" erkennen, die zweite Schicht komplexere Konstrukte wie ironische Wendungen. Wir benutzen dazu den Parameter `num_layers` der `nn.RNN`-Klasse.

Hier ist das vollständige Modell für Sentiment-Analyse:

```
class IMDBSentimentRNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size, num_layers,
                 output_size=2, dropout=0.1):
        super().__init__()

        # Embedding-Schicht: Wort-Indizes → Vektoren
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)

        # Gestapeltes RNN
        self.rnn = nn.RNN(
            embed_dim,
            hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout,
        )

        # Klassifikationsschicht
        self.fc = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # Embeddings berechnen
        embedded = self.embedding(x)  # [batch, seq_len, embed_dim]

        # RNN durchlaufen
        rnn_out, _ = self.rnn(embedded)  # [batch, seq_len, hidden_size]

        # Den letzten "echten" Hidden State finden (vor Padding)
        seq_lengths = (x != 0).sum(dim=1)
        indices = (seq_lengths - 1).clamp(min=0)

        batch_size = x.size(0)
```

```
        last_hidden = rnn_out[torch.arange(batch_size), indices]

        # Klassifikation
        output = self.dropout(last_hidden)
        return self.fc(output)
```

Der Parameter `padding_idx=0` im Embedding teilt PyTorch mit, dass Index 0 das Padding-Token ist. Die entsprechenden Embeddings werden auf Null gesetzt und nicht trainiert.

Die Zeilen zum Finden des "letzten echten Hidden States" sind wichtig: Weil Sequenzen unterschiedlich lang sind und mit Padding aufgefüllt wurden, wollen wir nicht den Hidden State nach dem letzten Padding-Token, sondern nach dem letzten echten Wort.

3.5 RNN Training und Evaluation

Training-Loop mit DataLoader

Das Training folgt dem bekannten Muster aus Kapitel 2: In jeder Epoche durchlaufen wir alle Batches der Trainingsdaten. Für jeden Batch setzen wir die Gradienten zurück (`zero_grad`), berechnen die Vorhersage (Forward Pass), ermitteln den Verlust, berechnen die Gradienten (Backward Pass mit `backward()`) und aktualisieren die Gewichte (`step()`). Nach jeder Epoche evaluieren wir das Modell auf den Validierungsdaten, um zu prüfen, ob es gut generalisiert.

```
# Konfiguration
MAX_LENGTH = 256
BATCH_SIZE = 64
EMBED_DIM = 200
HIDDEN_SIZE = 300
NUM_LAYERS = 2
DROPOUT = 0.1
LEARNING_RATE = 0.001
NUM_EPOCHS = 7
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Daten vorbereiten
train_labels = [ex["label"] for ex in dataset["train"]]
val_texts = [clean_text(ex["text"]) for ex in dataset["test"]]
val_labels = [ex["label"] for ex in dataset["test"]]

train_dataset = IMDBDataset(train_texts, train_labels, vocab, MAX_LENGTH)
val_dataset = IMDBDataset(val_texts, val_labels, vocab, MAX_LENGTH)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)

# Modell erstellen
model = IMDBSentimentRNN(
    vocab_size=len(vocab),
    embed_dim=EMBED_DIM,
    hidden_size=HIDDEN_SIZE,
```

```
    num_layers=NUM_LAYERS,
    dropout=DROPOUT
).to(DEVICE)

print(f"Parameter: {sum(p.numel() for p in model.parameters()):,}")
```

Die Trainings- und Evaluationsfunktionen:

```
def train_epoch(model, dataloader, loss_fn, optimizer, device):
    model.train()
    total_loss = 0
    correct = 0
    total = 0

    for texts, labels in dataloader:
        texts, labels = texts.to(device), labels.to(device)

        optimizer.zero_grad()
        logits = model(texts)
        loss = loss_fn(logits, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        _, predicted = torch.max(logits, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

    return total_loss / len(dataloader), 100 * correct / total


def eval_epoch(model, dataloader, loss_fn, device):
    model.eval()
    total_loss = 0
    correct = 0
    total = 0

    with torch.no_grad():
        for texts, labels in dataloader:
            texts, labels = texts.to(device), labels.to(device)
            logits = model(texts)
            loss = loss_fn(logits, labels)

            total_loss += loss.item()
            _, predicted = torch.max(logits, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    return total_loss / len(dataloader), 100 * correct / total
```

Die Trainingsschleife orchestriert das gesamte Training. Sie durchläuft die festgelegte Anzahl an Epochen und ruft in jeder Epoche die Trainings- und Evaluationsfunktionen auf. Nach jeder Epoche geben wir die Metriken aus, um den Fortschritt

zu überwachen. Achten Sie besonders auf die Validierungsgenauigkeit: Wenn sie nicht mehr steigt oder sogar sinkt während die Trainingsgenauigkeit weiter steigt, ist das ein Zeichen für Overfitting. D.h. das Modell lernt die Trainingsdaten auswendig, aber kann nicht mehr auf andere Daten generalisieren.

Und die Trainingsschleife:

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)

for epoch in range(NUM_EPOCHS):
    train_loss, train_acc = train_epoch(
        model, train_loader, loss_fn, optimizer, DEVICE
    )
    val_loss, val_acc = eval_epoch(model, val_loader, loss_fn, DEVICE)

    print(f"Epoch {epoch + 1}: "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%, "
          f"Val Acc: {val_acc:.2f}%")
```

Speichern und Laden von Modellen

Nach dem Training wollen Sie das Modell speichern, um es später wiederzuverwenden. Das ist wichtig, weil das Training oft Stunden oder sogar Tage dauern kann. Mit einem gespeicherten Modell können Sie es direkt für Vorhersagen nutzen, ohne erneut zu trainieren. Außerdem können Sie das Modell auf einem anderen Computer laden, es in eine Anwendung integrieren oder später das Training fortsetzen:

```
# Modell speichern
model_path = "imdb_rnn_model.pth"
vocab_path = "imdb_vocab.pth"

torch.save(model.state_dict(), model_path)
torch.save(vocab, vocab_path)

print(f"Modell gespeichert unter: {model_path}")
print(f"Vokabular gespeichert unter: {vocab_path}")
```

Das Laden funktioniert so:

```
# Vokabular laden
loaded_vocab = torch.load(vocab_path)

# Modell neu erstellen und Gewichte laden
loaded_model = IMDBSentimentRNN(
    vocab_size=len(loaded_vocab),
    embed_dim=EMBED_DIM,
    hidden_size=HIDDEN_SIZE,
    num_layers=NUM_LAYERS,
).to(DEVICE)

loaded_model.load_state_dict(torch.load(model_path, map_location=DEVICE))
loaded_model.eval()
```

Wichtig: Sie müssen das Modell mit derselben Architektur neu erstellen, bevor Sie die Gewichte laden. Die `.pth`-Datei enthält nur die Gewichte, nicht die Modellstruktur.

Inferenz: Vorhersagen mit dem trainierten Modell

Nachdem das Modell trainiert ist, können wir es für *Inferenz* verwenden. Das bedeutet, Vorhersagen für neue, bisher ungesehene Daten zu treffen. Bei der Inferenz ist es wichtig, das Modell in den Evaluierungsmodus zu versetzen (`model.eval()`). Zur Erinnerung: Im Evaluierungsmodus werden Dropout-Schichten deaktiviert und Batch Normalization verwendet die während des Trainings gelernten Statistiken statt der Batch-Statistiken. Außerdem verwenden wir `torch.no_grad()`, da wir keine Gradienten benötigen und so Speicher und Rechenzeit sparen.

```
def predict_sentiment(model, vocab, text, max_length=256, device=DEVICE):
    model.eval()

    # Text vorverarbeiten
    cleaned = clean_text(text)
    tokens = tokenize(cleaned)
    indices = [vocab.get(token, vocab["<UNK>"]) for token in tokens]

    # Padding/Truncation
    if len(indices) > max_length:
        indices = indices[:max_length]
    else:
        indices = indices + [vocab["<PAD>"]] * (max_length - len(indices))

    # Vorhersage
    x = torch.tensor([indices]).to(device)

    with torch.no_grad():
        logits = model(x)
        probs = torch.softmax(logits, dim=1)
        predicted = probs.argmax(dim=1).item()

    sentiment = "Positiv" if predicted == 1 else "Negativ"
    confidence = probs[0, predicted].item()

    return sentiment, confidence


# Beispiele testen
reviews = [
    "This movie was absolutely fantastic! Best film I've seen all year.",
    "Terrible waste of time. The plot made no sense and acting was awful.",
]

for review in reviews:
    sentiment, confidence = predict_sentiment(model, vocab, review)
    print(f"'{review[:50]}...' → {sentiment} ({confidence:.1%})")
```

Grenzen von unidirektionalen RNNs

Unser RNN liest Text von links nach rechts. Bei dem Satz “The movie was not ___” kennt das Netzwerk am Unterstrich bereits die Wörter “not”, aber es weiß nicht, was danach kommt. Für manche Aufgaben wäre es hilfreich, auch den nachfolgenden Kontext zu kennen.

Ein konkretes Beispiel: Im Satz “The bank by the river was flooded” ist das Wort “bank” mehrdeutig – es könnte eine Geldbank oder ein Flussufer bedeuten. Ein unidirektionales RNN müsste diese Entscheidung treffen, bevor es “by the river” sieht. Erst der nachfolgende Kontext (“river”, “flooded”) macht klar, dass es sich um ein Flussufer handelt.

Bidirektionale RNNs

Bidirektionale RNNs lösen dieses Problem, indem sie die Sequenz in *beide* Richtungen verarbeiten:

- **Vorwärts-RNN**: Liest von links nach rechts
- **Rückwärts-RNN**: Liest von rechts nach links

Technisch funktioniert das so: PyTorch erstellt intern zwei separate RNNs mit unabhängigen Gewichten. Das Vorwärts-RNN verarbeitet die Sequenz von Position 0 bis zum Ende, das Rückwärts-RNN von der letzten Position rückwärts bis zum Anfang. An jeder Position werden die beiden Hidden States zusammengefügt (konkateniert). Wenn der Hidden State jeder Richtung 128 Dimensionen hat, ist der kombinierte Hidden State 256 Dimensionen groß.

Die Hidden States beider Richtungen werden an jeder Position kombiniert (typischerweise durch Konkatenation, also werden die Tensoren einfach “aneinander geklebt”). Wir können dazu der Klasse `nn.RNN` einfach den Parameter `bidirectional` übergeben:

```
class BidirectionalRNNClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size, num_classes):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embed_dim)

        self.rnn = nn.RNN(
            input_size=embed_dim,
            hidden_size=hidden_size,
            num_layers=2,
            batch_first=True,
            bidirectional=True,  # Bidirektional aktivieren
        )

        # Ausgabeschicht: hidden_size * 2 wegen Bidirektionalität
        self.fc = nn.Linear(hidden_size * 2, num_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        output, hidden = self.rnn(embedded)
```

```
        # Letzten Zeitschritt verwenden
        last_output = output[:, -1, :]

        return self.fc(last_output)
```

Der einzige Unterschied im Code ist `bidirectional=True`. PyTorch kümmert sich um den Rest. Die Ausgabedimension verdoppelt sich, weil vorwärts und rückwärts Hidden States konkateniert werden.

3.6 Vanishing Gradients und GRU

Das Vanishing-Gradient-Problem

Bei langen Sequenzen haben einfache RNNs ein fundamentales Problem: Die Gradienten "verschwinden" während der Rückwärtspropagierung.

Erinnern Sie sich an die RNN-Formel: `h_t = tanh(W_h · h_{t-1} + W_x · x_t)`. Bei der Backpropagation durch die Zeit wird der Gradient bei jedem Schritt mit der Ableitung der tanh-Funktion und der Gewichtsmatrix W_h multipliziert. Die tanh-Ableitung ist maximal 1 (und oft viel kleiner). Wenn wir das über 100 Zeitschritte multiplizieren:

`0.5 × 0.5 × 0.5 × ... × 0.5 (100 mal) ≈ 0`

Der Gradient wird so klein, dass die frühen Zeitschritte praktisch nichts mehr lernen. Das Netzwerk "vergisst" den Anfang der Sequenz.

Warum Gradienten in RNNs verschwinden

Das Problem liegt in der Architektur: Bei jedem Zeitschritt wird der Hidden State *komplett neu berechnet*. Selbst wenn wichtige Information am Anfang der Sequenz war, wird sie durch die vielen Transformationen "verwässert".

Mathematisch ausgedrückt: Der Gradient fließt durch eine Kette von Matrixmultiplikationen und tanh-Ableitungen. Stellen Sie sich vor, Sie multiplizieren wiederholt mit einem Faktor: Wenn dieser Faktor kleiner als 1 ist (z.B. 0.9), wird das Ergebnis nach vielen Multiplikationen winzig klein (0.9^100 ≈ 0.00003). Genau das passiert mit den Gradienten, sie schrumpfen bei jedem Zeitschritt. Wenn der Faktor größer als 1 ist, explodieren die Gradienten stattdessen ins Unendliche (das sogenannte "Exploding Gradient Problem", das sich durch Gradient Clipping beheben lässt).

Gates als Lösung

Die Lösung sind *Gates* (Tore). Das sind lernbare Mechanismen, die kontrollieren, welche Information fließt und welche blockiert wird. Die Idee: Anstatt den Hidden State bei jedem Schritt komplett neu zu berechnen, kann das Netzwerk lernen, bestimmte Teile beizubehalten.

Konzeptionell funktioniert das so:

`neuer_state = gate * alter_state + (1 - gate) * kandidat_state`

Wenn das Gate nahe 1 ist, wird der alte State fast unverändert übernommen. Wenn es nahe 0 ist, wird der neue Kandidat verwendet (wie dieser berechnet wird siehe

unten). Das Entscheidende: Bei einem Gate nahe 1 fließt der Gradient direkt durch (multipliziert mit 1), ohne durch eine tanh-Funktion zu gehen. So kann Information über viele Zeitschritte erhalten bleiben.

Gates verwenden die Sigmoid-Funktion, deren Ausgabe zwischen 0 und 1 liegt: - Wert nahe 0: Information blockieren - Wert nahe 1: Information durchlassen

Zwei populäre gated Architekturen sind: - **LSTM** (Long Short-Term Memory): Hat drei Gates und einen separaten Cell State - **GRU** (Gated Recurrent Unit): Einfacher mit nur zwei Gates, aber oft genauso efffktiv

Mit der LSTM-Architekur werden wir uns in Kapitel 4 beschäftigen, zunächst zu den Gates in GRU.

GRU: Gated Recurrent Unit

GRU ist die einfachere der beiden gated Architekturen und für viele Anwendungen ausreichend. Sie hat zwei Gates, das Reset Gate und das Update Gate. Das Reset Gate verwenden wir für die Berechnung des Kandidat-State, das Update Gate dann für den finalen, neuen Hidden State

In den folgenden Formeln bezeichnet `[a, b]` die Konkatenation (Zusammenfügung) zweier Vektoren. Wenn `h_{t-1}` 128 Dimensionen hat und `x_t` 64 Dimensionen, dann hat `[h_{t-1}, x_t]` 192 Dimensionen. Die Gewichtsmatrix `W` hat entsprechend 192 Spalten.

Reset Gate (r): Bestimmt, wie viel vom vorherigen Hidden State “vergessen” werden soll

```
r_t = sigmoid(W_r · [h_{t-1}, x_t])
```

Die Gewichtsmatrix `W_r` transformiert den konkatenierten Vektor, und die Sigmoid-Funktion drückt jeden Wert in den Bereich 0 bis 1.

Update Gate (z): Bestimmt, wie viel vom alten Hidden State beibehalten wird

```
z_t = sigmoid(W_z · [h_{t-1}, x_t])
```

Kandidaten-Hidden-State: Hier wird der Reset Gate angewendet. Das * bedeutet elementweise Multiplikation. Jedes Element von `h_{t-1}` wird mit dem entsprechenden Gate-Wert multipliziert:

```
h'_t = tanh(W · [r_t * h_{t-1}, x_t])
```

Wenn `r_t` nahe 0 ist, wird der vorherige Hidden State “ausgeblendet” und der Kandidat basiert hauptsächlich auf der aktuellen Eingabe.

Finaler Hidden State: Die gewichtete Kombination aus altem und neuem State:

```
h_t = (1 - z_t) * h'_t + z_t * h_{t-1}
```

Die letzte Formel ist der Schlüssel: Wenn z_t nahe 1 ist, wird der alte Hidden State fast unverändert übernommen. Der Gradient kann dann ohne Abschwächung durch die Zeit fließen, weil die Information nicht durch eine tanh-Funktion gepresst wird.

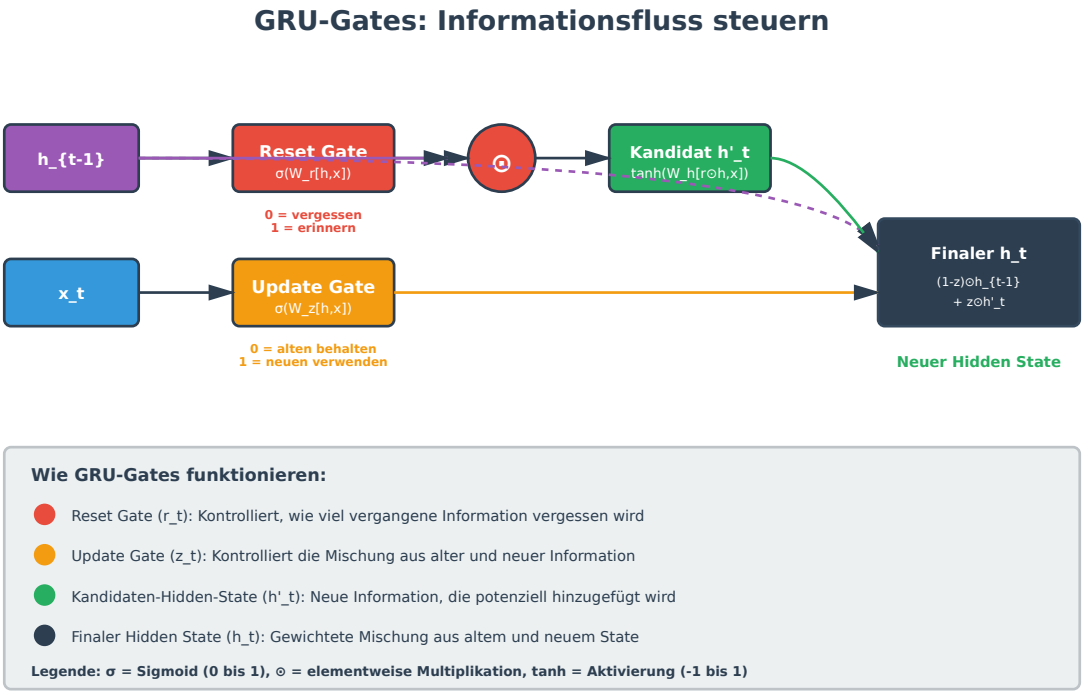

Abbildung 8: Die GRU-Zelle mit Reset Gate und Update Gate

Reset Gate und Update Gate verstehen

Reset Gate nahe 0: In der Formel `h'_t = tanh(W · [r_t * h_{t-1}, x_t])` wird `r_t * h_{t-1}` nahezu Null. Der Kandidaten-State wird dann fast ausschließlich aus der aktuellen Eingabe `x_t` berechnet. Das ist nützlich, wenn ein neues Thema beginnt und die alte Information irrelevant wird, zum Beispiel bei einem Satzwechsel.

Reset Gate nahe 1: Der vorherige Hidden State `h_{t-1}` fließt vollständig in die Berechnung des Kandidaten ein. Das ist nützlich, wenn das aktuelle Wort stark vom vorherigen Kontext abhängt, zum Beispiel bei zusammengesetzten Ausdrücken wie “nicht gut”.

Update Gate nahe 0: In der Formel `h_t = (1 - z_t) * h'_t + z_t * h_{t-1}` dominiert der erste Term. Der neue Kandidaten-State `h'_t` wird fast vollständig übernommen. Das passiert, wenn neue, wichtige Information aufgenommen werden soll.

Update Gate nahe 1: Der zweite Term dominiert, der alte Hidden State `h_{t-1}` wird fast unverändert weitergegeben. Das ist der Schlüssel zur Lösung des Vanishing-Gradient-Problems: Der Gradient kann direkt durch diesen “Bypass” fließen, ohne durch eine tanh-Funktion abgeschwächt zu werden. So können wichtige Informationen über viele Zeitschritte erhalten bleiben.

GRU für IMDB-Sentiment-Analyse

Das Schöne an PyTorchs Abstraktion: Um von RNN zu GRU zu wechseln, müssen Sie nur eine Zeile ändern:

```python
class IMDBSentimentGRU(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size, num_layers,
                 output_size=2, dropout=0.1):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)

        # Nur diese Zeile ändert sich: nn.RNN → nn.GRU
        self.rnn = nn.GRU(
            embed_dim,
            hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout,
        )

        self.fc = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        embedded = self.embedding(x)
        rnn_out, _ = self.rnn(embedded)

        seq_lengths = (x != 0).sum(dim=1)
        indices = (seq_lengths - 1).clamp(min=0)

        batch_size = x.size(0)
        last_hidden = rnn_out[torch.arange(batch_size, device=x.device), indices]

        output = self.dropout(last_hidden)
        return self.fc(output)
```

Der Rest des Codes, Datenverarbeitung, Training, Evaluation, bleibt identisch. GRU hat mehr Parameter als ein einfaches RNN (drei Gewichtsmatrizen statt einer), aber die zusätzliche Kapazität zahlt sich bei langen Sequenzen aus.

Bidirektionale GRU

Für noch bessere Ergebnisse können Sie GRU mit Bidirektionalität kombinieren:

```python
self.rnn = nn.GRU(
    embed_dim,
    hidden_size,
    num_layers=num_layers,
    batch_first=True,
    dropout=dropout,
    bidirectional=True,  # Bidirektional aktivieren
)

# Die Ausgabeschicht muss angepasst werden
self.fc = nn.Linear(hidden_size * 2, output_size)  # * 2 wegen Bidirektionalität
```

Mit bidirektionaler GRU können Sie auf dem IMDB-Datensatz Genauigkeiten von

85-90% erreichen. Eine deutliche Verbesserung gegenüber dem einfachen unidirektionalen RNN.

Zusammenfassung

In diesem Kapitel haben Sie zwei spezialisierte Netzwerkarchitekturen kennengelernt:

Convolutional Neural Networks (CNNs) für Bildverarbeitung: - Convolution-Operationen erkennen lokale Muster unabhängig von ihrer Position - Pooling reduziert die räumliche Größe und erhöht die Robustheit - Typische Architektur: Conv → BatchNorm → ReLU → Pool → (wiederholen) → Linear - Data Augmentation hilft, mit begrenzten Trainingsdaten bessere Ergebnisse zu erzielen

Recurrent Neural Networks (RNNs) für sequenzielle Daten: - Embeddings wandeln Wörter in dichte Vektoren um - Der Hidden State überträgt Information von einem Zeitschritt zum nächsten - Gestapelte RNNs lernen hierarchische Repräsentationen - Bidirektionale RNNs nutzen Kontext aus beiden Richtungen

Das Vanishing-Gradient-Problem begrenzt einfache RNNs bei langen Sequenzen: - Gradienten schrumpfen exponentiell über die Zeit - **GRU** (Gated Recurrent Unit) löst das Problem mit lernbaren Gates - Gates kontrollieren den Informationsfluss und erlauben Gradienten, ungehindert zu fließen

Im nächsten Kapitel werden wir LSTM (Long Short-Term Memory) kennenlernen, eine noch mächtigere gated Architektur, und den Attention-Mechanismus einführen, der die Grundlage für moderne Sprachmodelle wie GPT und BERT bildet.

Kapitel 4: LSTM, Encoder-Decoder und Attention

Im vorherigen Kapitel haben Sie Recurrent Neural Networks (RNNs) und GRUs kennengelernt, die sequenzielle Daten wie Text verarbeiten können. In diesem Kapitel gehen wir noch einen Schritt weiter: Sie lernen LSTM kennen, eine noch mächtigere RNN-Variante, und die Encoder-Decoder-Architektur für Aufgaben wie maschinelle Übersetzung. Schließlich führen wir den revolutionären Attention-Mechanismus ein, der die Grundlage für moderne Sprachmodelle wie GPT und BERT bildet.

4.1 Variable Sequenzlängen: Packing und Unpacking

Das Padding-Problem

In Kapitel 3 haben wir alle Sequenzen auf dieselbe Länge gebracht, indem wir kürzere Sequenzen mit <PAD>-Tokens aufgefüllt haben. Stellen Sie sich einen Batch mit Sequenzen der Längen 5, 12 und 3 vor. Alle werden auf Länge 12 gepaddet:

```
Sequenz 1: [Wort1, Wort2, Wort3, Wort4, Wort5, PAD, PAD, PAD, PAD, PAD, PAD, PAD]
Sequenz 2: [Wort1, Wort2, ..., Wort12]
Sequenz 3: [Wort1, Wort2, Wort3, PAD, PAD, PAD, PAD, PAD, PAD, PAD, PAD, PAD]
```

Das funktioniert, hat aber mehrere Nachteile:

- **Verschwendete Rechenzeit**: Das RNN verarbeitet jeden Padding-Token, obwohl dieser keine Information enthält
- **Verschwendeter Speicher**: Padding-Tokens belegen Platz im Tensor
- **Verfälschte Hidden States**: Das RNN aktualisiert seinen Hidden State auch bei Padding-Tokens

Warum Padding Hidden States beeinflusst

Das Problem mit den Hidden States verdient besondere Aufmerksamkeit. Ein RNN berechnet bei jedem Zeitschritt einen neuen Hidden State:

```
h_t = tanh(W_h · h_{t-1} + W_x · x_t + b)
```

Auch wenn `x_t` ein Padding-Token ist (typischerweise als Null-Vektor repräsentiert), fließt `h_{t-1}` weiterhin in die Berechnung ein. Der Hidden State verändert sich also auch bei Padding-Tokens. Bei bidirektionalen RNNs wird dieses Problem noch verstärkt, weil Padding-Tokens sowohl in der Vorwärts- als auch in der Rückwärtsrichtung verarbeitet werden.

Das Ergebnis: Die finalen Hidden States von kurzen Sequenzen sind weniger aussagekräftig, weil sie durch viele "leere" Berechnungen verwässert wurden.

Packed Sequences in PyTorch

PyTorch bietet eine elegante Lösung: *Packed Sequences*. Die Idee ist, die Padding-Tokens aus dem Batch zu entfernen, bevor das RNN sie verarbeitet. Das RNN sieht dann nur die echten Tokens.

Der Ablauf ist: 1. `pack_padded_sequence`: Komprimiert den Batch und entfernt Padding 2. RNN-Verarbeitung: Nur echte Tokens werden verarbeitet 3. `pad_packed_sequence`: Stellt das ursprüngliche Format wieder her

Hier ist ein vollständiges Beispiel. Zunächst erzeugen wir Tensoren mit Padding mit der Funktion `pad_sequence()`:

```
import torch
import torch.nn as nn

torch.manual_seed(42)

# Beispiel-Batch: 3 Sequenzen mit unterschiedlichen Längen
sequences = [
    torch.tensor([1, 2, 3, 4, 5]),      # Länge 5
    torch.tensor([6, 7, 8]),             # Länge 3
    torch.tensor([9, 10, 11, 12, 13, 14]),  # Länge 6
]

# Sequenzen padden
lengths = torch.tensor([len(seq) for seq in sequences])
padded = nn.utils.rnn.pad_sequence(sequences, batch_first=True)

print("Gepaddete Sequenzen:")
print(padded)
print(f"Shape: {padded.shape}")
print(f"Längen: {lengths}")
```

Die Ausgabe zeigt den gepadden Tensor:

```
Gepaddete Sequenzen:
tensor([[ 1,  2,  3,  4,  5,  0],
        [ 6,  7,  8,  0,  0,  0],
        [ 9, 10, 11, 12, 13, 14]])
Shape: torch.Size([3, 6])
Längen: tensor([5, 3, 6])
```

Nun können wir die Sequenzen packen und durch ein RNN schicken:

```
# Embedding-Layer erstellen
vocab_size = 20
embed_dim = 4
embedding = nn.Embedding(vocab_size, embed_dim)

# Embeddings berechnen
embedded = embedding(padded)
```

```
print(f"Embedded Shape: {embedded.shape}")

# Sequenzen packen
packed = nn.utils.rnn.pack_padded_sequence(
    embedded, lengths, batch_first=True, enforce_sorted=False
)

print(f"\nPacked Sequences Data Shape: {packed.data.shape}")
print(f"Batch Sizes: {packed.batch_sizes}")
```

Der packed-Tensor hat eine besondere Struktur. Die data-Komponente enthält alle echten Tokens ohne Padding. Die batch_sizes geben an, wie viele Elemente pro Zeitschritt vorhanden sind.

```
Embedded Shape: torch.Size([3, 6, 4])

Packed Sequences Data Shape: torch.Size([14, 4])
Batch Sizes: tensor([3, 3, 3, 2, 2, 1])
```

Die 14 in packed.data.shape kommt von den echten Längen der Sequenzen: 5 + 3 + 6 = 14 Tokens. Die batch_sizes zeigen, dass am ersten Zeitschritt alle 3 Sequenzen ein Token haben, dann wieder alle 3, und so weiter, bis am Ende nur noch die längste Sequenz übrig ist.

RNN mit Packed Sequences

Jetzt verarbeiten wir die gepackten Sequenzen mit einem bidirektionalen RNN. Das Modell erkennt dabei automatisch, ob es sich bei den Eingabetensoren um gepackte oder normale Tensoren handelt:

```
# RNN erstellen
hidden_size = 8
rnn = nn.RNN(embed_dim, hidden_size, batch_first=True, bidirectional=True)

# Packed Sequences verarbeiten
packed_output, hidden = rnn(packed)

print(f"Packed Output Data Shape: {packed_output.data.shape}")
print(f"Hidden State Shape: {hidden.shape}")

# Wieder in gepaddes Format bringen
output, output_lengths = nn.utils.rnn.pad_packed_sequence(
    packed_output, batch_first=True
)

print(f"\nUnpacked Output Shape: {output.shape}")
print(f"Output Lengths: {output_lengths}")
```

```
Packed Output Data Shape: torch.Size([14, 16])
Hidden State Shape: torch.Size([2, 3, 8])

Unpacked Output Shape: torch.Size([3, 6, 16])
Output Lengths: tensor([5, 3, 6])
```

Die Ausgabe hat 16 Dimensionen (8 × 2 wegen Bidirektionalität). Nach dem Entpacken haben wir wieder einen regulären Tensor mit Padding.

Den letzten echten Hidden State extrahieren

Ein wichtiger Punkt: Bei Packed Sequences können wir nicht einfach `output[:, -1, :]` verwenden, um den letzten Hidden State zu bekommen. Bei gepackten Sequenzen ist die letzte Position unterschiedlich für jede Sequenz. Bisher haben wir das ignoriert und einfach den Hidden State nach dem letzten Token genommen, auch wenn die Sequenz mit Padding Tokens endete. Nun werden für diese Padding Tokens keine Hidden States mehr berechnet. Wir müssen die echten Längen berücksichtigen:

```
# Den letzten echten Hidden State für jede Sequenz extrahieren
seq_idx = torch.arange(lengths.size(0))
last_hidden = output[seq_idx, lengths - 1]
print(f"Last Hidden States Shape: {last_hidden.shape}")
```

Diese Indizierung greift für jede Sequenz den Output an der Position `länge - 1` ab, also genau nach dem letzten echten Token.

Praktische Tipps für Packed Sequences

Einige wichtige Hinweise für die Arbeit mit Packed Sequences:

Sortierung: Standardmäßig erwartet `pack_padded_sequence`, dass die Sequenzen nach Länge sortiert sind (längste zuerst). Mit `enforce_sorted=False` können Sie unsortierte Batches verwenden. Das ist bequemer, aber minimal langsamer.

CPU-Längen: Die `lengths` müssen auf der CPU liegen, auch wenn die Sequenzen auf der GPU sind:

```
# Richtig:
packed = nn.utils.rnn.pack_padded_sequence(
    embedded.cuda(), lengths.cpu(), batch_first=True, enforce_sorted=False
)
```

4.2 LSTM: Long Short-Term Memory

Motivation: Vanishing Gradients

In Kapitel 3 haben wir das Vanishing-Gradient-Problem kennengelernt: Bei langen Sequenzen schrumpfen die Gradienten exponentiell, sodass das RNN den Anfang der Sequenz nicht effektiv lernen kann. GRUs lösen dieses Problem teilweise mit ihren zwei Gates (Reset und Update).

LSTM (Long Short-Term Memory) geht noch einen Schritt weiter. Es wurde bereits 1997 von Hochreiter und Schmidhuber entwickelt und ist bis heute eine der wichtigsten RNN-Architekturen. LSTMs führen einen separaten *Cell State* ein, der wie eine “Gedächtnis-Autobahn” funktioniert.

Cell State: Die "Gedächtnis-Autobahn"

Der entscheidende Unterschied zwischen GRU und LSTM ist der Cell State (Zellzustand). Während bei GRU und einfachen RNNs nur der Hidden State von Zeitschritt zu Zeitschritt weitergegeben wird, hat LSTM zwei parallel laufende Zustände:

- **Hidden State (h)**: Wie bei anderen RNNs, die "sichtbare" Ausgabe
- **Cell State (c)**: Ein separater Speicher, der Information über lange Zeiträume bewahren kann

Der Cell State ist der Schlüssel zur Lösung des Vanishing-Gradient-Problems. Bei einem einfachen RNN wird der Hidden State bei jedem Zeitschritt mit einer Gewichtsmatrix multipliziert: `h_t = tanh(W · h_{t-1} + ...)`. Wenn wir Gradienten über viele Zeitschritte zurückpropagieren, werden diese Matrixmultiplikationen aneinandergereiht, was zu exponentiell schrumpfenden oder explodierenden Gradienten führt.

Der Cell State in LSTM hingegen wird durch eine *additive* Operation aktualisiert: `c_t = f_t * c_{t-1} + i_t * g_t`. Das `+` ist entscheidend. Beim Backpropagation fließt der Gradient durch Addition unverändert zurück, denn die Ableitung von `a + b` nach `a` ist einfach 1.

Betrachten Sie ein konkretes Beispiel: Wenn das Forget Gate `f_t ≈ 1` ist (Information behalten), dann gilt `c_t ≈ c_{t-1} + i_t * g_t`. Der Gradient bezüglich `c_{t-1}` ist dann nahezu 1. Selbst nach 100 Zeitschritten kann der Gradient ohne signifikante Abschwächung vom Ende zum Anfang der Sequenz fließen, da wir hauptsächlich addieren statt wiederholt zu multiplizieren.

Stellen Sie sich den Cell State als eine Autobahn vor: Information kann ungebremst von einem Ende zum anderen fließen. Gates kontrollieren, was auf die Autobahn aufgeladen, was abgeladen und was ausgegeben wird.

LSTM-Gates: Forget, Input, Output

LSTM verwendet drei Gates, die jeweils eine Sigmoid-Aktivierung haben (Werte zwischen 0 und 1):

Forget Gate (f): Entscheidet, welche Information aus dem Cell State "vergessen" werden soll. - Wert nahe 0: Information löschen - Wert nahe 1: Information behalten

Input Gate (i): Entscheidet, welche neue Information in den Cell State geschrieben wird. - Wert nahe 0: Neue Information ignorieren - Wert nahe 1: Neue Information aufnehmen

Output Gate (o): Entscheidet, welcher Teil des Cell State als Hidden State ausgegeben wird. - Wert nahe 0: Information intern behalten - Wert nahe 1: Information nach außen geben

LSTM-Formeln verstehen

Alle drei Gates werden aus dem aktuellen Input und dem vorherigen Hidden State berechnet:

```
f_t = sigmoid(W_f · [h_{t-1}, x_t] + b_f)   # Forget Gate
i_t = sigmoid(W_i · [h_{t-1}, x_t] + b_i)   # Input Gate
o_t = sigmoid(W_o · [h_{t-1}, x_t] + b_o)   # Output Gate
```

Zusätzlich wird ein Kandidaten-Cell-State berechnet:

```
g_t = tanh(W_g · [h_{t-1}, x_t] + b_g)      # Candidate Cell State
```

Jetzt kommt die Aktualisierung des Cell State. Hier ist die "Magie" von LSTM:

```
c_t = f_t * c_{t-1} + i_t * g_t
```

Der neue Cell State ist eine Kombination aus: - Dem alten Cell State, gefiltert durch das Forget Gate - Dem Kandidaten, gefiltert durch das Input Gate

Das * bedeutet elementweise Multiplikation. Der Gradient kann durch das + direkt fließen, ohne abgeschwächt zu werden.

Schließlich wird der Hidden State berechnet:

```
h_t = o_t * tanh(c_t)
```

Der Cell State wird durch tanh normalisiert und dann vom Output Gate gefiltert.

Gate-Verhalten interpretieren

Betrachten wir einige Szenarien:

Forget Gate ≈ 1, Input Gate ≈ 0: Das LSTM behält den alten Cell State und ignoriert neue Information. Das ist nützlich, wenn irrelevante Wörter verarbeitet werden, aber wichtige Information aus der Vergangenheit erhalten bleiben soll. Beispiel: Das Subjekt "Der Hund" muss über viele Wörter hinweg gespeichert werden, um später das Verb korrekt zu konjugieren.

Forget Gate ≈ 0, Input Gate ≈ 1: Das LSTM ersetzt den alten Cell State mit neuer Information. Das passiert bei Themenwechseln oder wenn komplett neue Information wichtiger ist. Beispiel: Nach einem Punkt beginnt ein neuer Satz, die alte Information wird unwichtig.

Output Gate ≈ 1: Das LSTM gibt den aktuellen Cell State als Hidden State aus. Das ist bei Positionen wichtig, die für die Vorhersage relevant sind.

Output Gate ≈ 0: Das LSTM "versteckt" den Cell State. Die Information wird intern weiterverarbeitet, aber noch nicht ausgegeben. Das ist bei Zwischenschritten nützlich, wenn das Netzwerk noch nicht bereit ist, eine Entscheidung zu treffen.

LSTM vs. GRU: Wann welche Architektur?

Hier ein Vergleich der beiden Architekturen:

Eigenschaft	LSTM	GRU
Gates	3 (Forget, Input, Output)	2 (Reset, Update)
Zustände	2 (Hidden, Cell)	1 (Hidden)
Parameter	Mehr	Weniger
Geschwindigkeit	Langsamer	Schneller
Performance	Oft besser bei komplexen Aufgaben	Oft ähnlich unabhängig von Komplexität

Wann sollten Sie LSTM verwenden?

- **Lange Sequenzen** (100+ Zeitschritte): LSTMs können langfristige Abhängigkeiten oft besser lernen
- **Komplexe Abhängigkeiten**: Wenn mehrere Informationen parallel gespeichert werden müssen
- **Ausreichend Rechenkapazität**: LSTM ist langsamer, aber mächtiger
- **Empirische Verbesserung**: Wenn Benchmarks zeigen, dass LSTM besser abschneidet

In der Praxis ist der Unterschied oft gering. GRU ist ein guter Ausgangspunkt, und LSTM lohnt sich, wenn Sie an die Grenzen von GRU stoßen.

LSTM in PyTorch

Die Verwendung von LSTM in PyTorch ist fast identisch zu GRU:

```
import torch
import torch.nn as nn

torch.manual_seed(42)

class StackedLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers=2):
        super().__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size

        self.rnn = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
        )

    def forward(self, x):
        output, (h_final, c_final) = self.rnn(x)
        return output, h_final, c_final


# Beispiel
model = StackedLSTM(input_size=4, hidden_size=8, num_layers=2)
x = torch.randn(2, 20, 4)  # Batch=2, Sequenzlänge=20, Features=4
output, final_hidden, final_cell = model(x)

print(f"Output Shape: {output.shape}")
print(f"Final Hidden Shape: {final_hidden.shape}")
print(f"Final Cell Shape: {final_cell.shape}")
```

```
Output Shape: torch.Size([2, 20, 8])
Final Hidden Shape: torch.Size([2, 2, 8])
Final Cell Shape: torch.Size([2, 2, 8])
```

Der einzige Unterschied zu GRU: LSTM gibt zusätzlich zum Hidden State auch den Cell State zurück. Bei mehreren Layern enthält `final_hidden` den Hidden State

jeder Schicht, ebenso `final_cell`.

4.3 Sequence-to-Sequence Probleme

Was sind Seq2Seq-Aufgaben?

Bisher haben wir RNNs für Klassifikation verwendet: Eine Sequenz geht hinein, eine einzelne Vorhersage kommt heraus. Aber viele interessante Aufgaben erfordern, dass die Ausgabe selbst eine Sequenz ist:

- **Maschinelle Übersetzung**: "Hello, how are you?" → "Hallo, wie geht es dir?"
- **Textzusammenfassung**: Langer Artikel → Kurze Zusammenfassung
- **Chatbots**: Benutzeranfrage → Bot-Antwort
- **Spracherkennung**: Audiosignal → Transkription

Diese Aufgaben nennt man *Sequence-to-Sequence* (Seq2Seq). Die zentrale Herausforderung: Eingabe und Ausgabe haben unterschiedliche Längen, und die Länge der Ausgabe ist nicht im Voraus bekannt.

Die zentrale Herausforderung

Betrachten Sie das Beispiel der Übersetzung von "I love machine learning" ins Französische:

- Englisch: 4 Wörter
- Französisch: "J'adore l'apprentissage automatique" (3 Wörter)

Die Wortanzahl ändert sich, und auch die Wortstellung kann anders sein. Jedes Ausgabewort hängt von der *gesamten* Eingabesequenz ab, nicht nur von einem einzelnen Eingabewort.

Ein einfaches RNN, das Wort für Wort übersetzt, funktioniert hier nicht. Wir brauchen eine Architektur, die erst die gesamte Eingabe "versteht" und dann eine variable Ausgabe erzeugt.

Encoder-Decoder-Architektur

Die Lösung ist die *Encoder-Decoder-Architektur*, bestehend aus zwei separaten neuronalen Netzen:

Encoder: Liest die Eingabesequenz und komprimiert sie in einen Vektor fester Größe. Dieser Vektor soll die "Bedeutung" der gesamten Eingabe erfassen.

Decoder: Nimmt den Vektor vom Encoder und generiert daraus Schritt für Schritt die Ausgabesequenz.

```
Eingabe: "I love machine learning"
    ↓
[Encoder] → Kontext-Vektor (feste Größe, z.B. 256 Dimensionen)
    ↓
[Decoder] → "J'adore" → "l'apprentissage" → "automatique" → <EOS>
```

Encoder: Input verarbeiten

Der Encoder ist ein RNN (oder LSTM/GRU), das die Eingabesequenz Wort für Wort liest. Nach dem letzten Wort enthält der Hidden State eine komprimierte Repräsentation der gesamten Eingabe:

```
import torch
import torch.nn as nn

torch.manual_seed(42)

class Encoder(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers=2):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size, padding_idx=0)
        self.lstm = nn.LSTM(
            embed_size, hidden_size, num_layers=num_layers, batch_first=True
        )

    def forward(self, x):
        embedded = self.embedding(x)
        outputs, (hidden, cell) = self.lstm(embedded)
        return hidden, cell


# Beispiel
encoder = Encoder(vocab_size=1000, embed_size=64, hidden_size=128)
src = torch.randint(1, 1000, (4, 20))  # Batch=4, Sequenzlänge=20
hidden, cell = encoder(src)

print(f"Encoder Hidden Shape: {hidden.shape}")
print(f"Encoder Cell Shape: {cell.shape}")
```

```
Encoder Hidden Shape: torch.Size([2, 4, 128])
Encoder Cell Shape: torch.Size([2, 4, 128])
```

Die finalen Hidden und Cell States werden an den Decoder übergeben. Sie enthalten die gesamte Information der Eingabesequenz, komprimiert in Vektoren mit 128 Dimensionen pro Layer. Der Decoder nutzt auch ein LSTM, so dass wir dort den Hidden und Cell State aus dem letzten Layer des Encoders als Eingabe übergeben werden.

Decoder: Output generieren

Der Decoder ist ebenfalls ein RNN, aber er arbeitet anders:

1. Er wird mit den finalen States des Encoders initialisiert
2. Er generiert ein Wort nach dem anderen
3. Jedes generierte Wort wird als Eingabe für den nächsten Schritt verwendet
4. Er stoppt, wenn er ein spezielles <EOS>-Token (End of Sequence) generiert

```
class Decoder(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size):
        super().__init__()
```

```
        self.embedding = nn.Embedding(vocab_size, embed_size, padding_idx=0)
        self.lstm = nn.LSTM(embed_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden, cell):
        embedded = self.embedding(x)
        output, (hidden, cell) = self.lstm(embedded, (hidden, cell))
        prediction = self.fc(output)
        return prediction, hidden, cell
```

Der Decoder ist *autoregressiv*: Seine eigene Ausgabe wird zur Eingabe des nächsten Schritts. Beim ersten Schritt erhält er ein spezielles <SOS>-Token (Start of Sequence):

```
# Decoder initialisieren
decoder = Decoder(vocab_size=1000, embed_size=64, hidden_size=128)

# Mit Encoder-States starten (nur letzter Layer)
encoder_hidden = hidden[-1:, :, :]  # [1, batch, hidden]
encoder_cell = cell[-1:, :, :]

# Erstes Token ist <SOS> (angenommen Index 2)
decoder_input = torch.full((4, 1), fill_value=2)  # Batch=4

# Generierung simulieren
for step in range(5):
    prediction, encoder_hidden, encoder_cell = decoder(
        decoder_input, encoder_hidden, encoder_cell
    )
    # Nächstes Token ist die Vorhersage
    decoder_input = prediction.argmax(dim=-1)
    print(f"Schritt {step + 1}: Prediction Shape {prediction.shape}")
```

Der Encoder hat 2 Layer (`num_layers=2`), daher haben die Hidden und Cell States die Form `[2, batch, hidden]`. Der Decoder in diesem Beispiel hat nur 1 Layer, daher benötigt er States der Form `[1, batch, hidden]`. Mit `hidden[-1:]` bzw. `cell[-1:]` extrahieren wir nur den letzten Layer (Index -1), behalten aber die erste Dimension mit `:`, sodass die Form `[1, batch, hidden]` entsteht. Würden wir `hidden[-1]` ohne Doppelpunkt verwenden, hätten wir `[batch, hidden]` und müssten die Dimension manuell mit `unsqueeze(0)` wieder hinzufügen.

Der Information Bottleneck

Die Encoder-Decoder-Architektur hat ein fundamentales Problem: Der gesamte Inhalt der Eingabesequenz muss in einen Vektor fester Größe passen. Bei einem 50 Wörter langen Satz, der in einen 256-dimensionalen Vektor komprimiert wird, geht unweigerlich Information verloren.

Stellen Sie sich vor, Sie müssten den Inhalt eines ganzen Buches in einem einzigen Satz zusammenfassen. Genau das verlangt der Encoder-Decoder, bei längeren Sequenzen überfordert diese Anforderung das Modell.

Die Folgen: - Performance sinkt bei längeren Sequenzen - Details vom Anfang der Eingabe werden “vergessen” - Der Decoder hat keinen direkten Zugriff auf einzelne

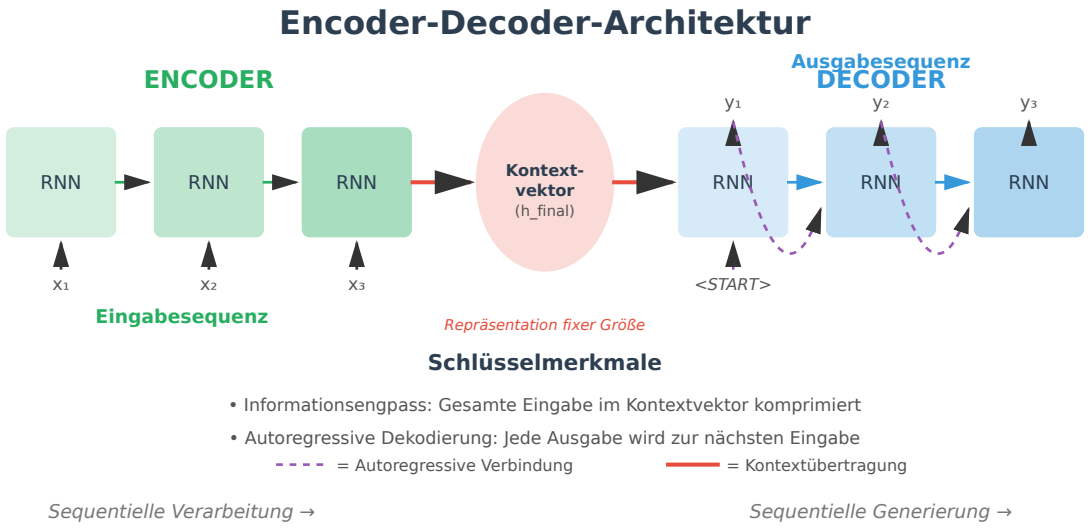

Abbildung 9: Die Encoder-Decoder-Architektur: Der Encoder verarbeitet die Eingabesequenz und übergibt seinen finalen Hidden State an den Decoder, der daraus die Ausgabesequenz generiert.

Eingabepositionen

Die Lösung für dieses Problem ist der Attention-Mechanismus, den wir in Abschnitt 4.5 kennenlernen werden. Aber zunächst müssen wir verstehen, wie man Encoder-Decoder-Modelle überhaupt trainiert.

4.4 Teacher Forcing

Motivation: Lernen ohne gelernt zu haben

Beim Training eines Encoder-Decoder-Modells gibt es ein Henne-Ei-Problem: Der Decoder soll lernen, korrekte Ausgaben zu produzieren, aber seine Ausgaben werden als Eingaben für den nächsten Schritt verwendet. Am Anfang des Trainings sind die Vorhersagen des Decoders zufällig und meist falsch.

Das führt zu einem Teufelskreis: 1. Decoder macht falsche Vorhersage 2. Falsche Vorhersage wird als nächste Eingabe verwendet 3. Nächste Vorhersage ist noch falscher 4. Fehler akkumulieren sich

Nach wenigen Schritten ist die Ausgabe komplett sinnlos, und das Modell hat keine Chance zu lernen.

Was ist Teacher Forcing?

Teacher Forcing löst dieses Problem elegant: Während des Trainings verwenden wir nicht die Vorhersagen des Decoders als nächste Eingabe, sondern die echten Ziel-Tokens aus den Trainingsdaten. Der “Lehrer” (die Ground Truth) “zwingt” den Decoder, von den korrekten Vorgängern zu lernen.

```
Ohne Teacher Forcing:
Eingabe: <SOS> → Decoder → "Je" (falsch, sollte "J'adore" sein)
```

```
          ↓
        "Je" → Decoder → "suis" (noch falscher)
          ↓
        ...

Mit Teacher Forcing:
Eingabe: <SOS> → Decoder → Vorhersage (egal)
          ↓                      ↓
Ziel: "J'adore" → Decoder → Vorhersage
          ↓                      ↓
Ziel: "l'apprentissage" → Decoder → ...
```

Der Decoder lernt: "Wenn ich 'J'adore' als Eingabe bekomme, sollte ich 'l'apprentissage' ausgeben." Das ist viel effektiver als vom eigenen Chaos zu lernen.

Training vs. Inferenz

Wichtig: Teacher Forcing wird nur während des Trainings verwendet. Bei der Inferenz (wenn das Modell auf echten Daten Vorhersagen macht) gibt es keine Ground Truth. Dann muss der Decoder seine eigenen Vorhersagen verwenden.

Oft verwendet man einen Kompromiss: Mit einer gewissen Wahrscheinlichkeit (z.B. 50%) wird Teacher Forcing verwendet, sonst die eigene Vorhersage. Das verhindert, dass das Modell zu abhängig von perfekten Eingaben wird.

Hier ist eine vollständige Seq2Seq-Klasse mit Teacher Forcing. Die Encoder- und Decoder-Klassen entsprechen dabei unseren obigen Implementierungen:

```python
import random

import torch
import torch.nn as nn

torch.manual_seed(42)
random.seed(42)

VOCAB_SIZE = 1000
EMBED_SIZE = 64
HIDDEN_SIZE = 128

# Geteiltes Embedding für Encoder und Decoder (optional, aber häufig)
embedding = nn.Embedding(VOCAB_SIZE, EMBED_SIZE, padding_idx=0)


class Encoder(nn.Module):
    def __init__(self, embedding, hidden_size):
        super().__init__()
        self.embedding = embedding
        self.lstm = nn.LSTM(
            embedding.embedding_dim, hidden_size, batch_first=True
        )
```

```
    def forward(self, x):
        embedded = self.embedding(x)
        outputs, (hidden, cell) = self.lstm(embedded)
        return hidden, cell


class Decoder(nn.Module):
    def __init__(self, embedding, hidden_size):
        super().__init__()
        self.embedding = embedding
        self.lstm = nn.LSTM(
            embedding.embedding_dim, hidden_size, batch_first=True
        )
        self.fc = nn.Linear(hidden_size, embedding.num_embeddings)

    def forward(self, x, hidden, cell):
        embedded = self.embedding(x)
        output, (hidden, cell) = self.lstm(embedded, (hidden, cell))
        prediction = self.fc(output)
        return prediction, hidden, cell


class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, src, trg, teacher_forcing_ratio=0.5):
        batch_size = src.size(0)
        trg_len = trg.size(1)

        # Tensor für Decoder-Ausgaben
        outputs = torch.zeros(batch_size, trg_len, dtype=torch.long)

        # Encoder durchlaufen
        hidden, cell = self.encoder(src)

        # Erstes Decoder-Token ist <SOS> (Index 2)
        decoder_input = torch.full((batch_size, 1), fill_value=2)

        # Sequenz generieren
        for t in range(trg_len):
            output, hidden, cell = self.decoder(decoder_input, hidden, cell)

            # Teacher Forcing: Ground Truth oder eigene Vorhersage?
            teacher_force = random.random() < teacher_forcing_ratio
            if teacher_force:
                decoder_input = trg[:, t:t+1]  # Echtes Token
            else:
                decoder_input = output.argmax(dim=-1)  # Eigene Vorhersage
```

```
            outputs[:, t] = decoder_input.squeeze(1)

        return outputs
```

Gradient Clipping

RNNs, besonders bei längeren Sequenzen, können unter *explodierenden Gradienten* leiden, dem Gegenteil von Vanishing Gradients. Das Problem entsteht durch die wiederholte Multiplikation mit Gewichtsmatrizen während der Backpropagation. Wenn diese Multiplikationen den Gradienten im Durchschnitt vergrößern statt verkleinern, wachsen die Gradienten bei jedem Zeitschritt exponentiell.

Stellen Sie sich vor, bei jedem Zeitschritt wird der Gradient mit einem Faktor von 1,5 multipliziert. Nach 50 Zeitschritten ist der Gradient um den Faktor 1,5^50 ≈ 637.621.500.000.000.000 gewachsen. Solche astronomischen Werte führen zu numerischer Instabilität: Die Gewichtsupdates werden so groß, dass das Modell "explodiert" und die Gewichte zu NaN (Not a Number) werden oder wild oszillieren.

Die Lösung ist *Gradient Clipping*: Gradienten, die eine bestimmten Wert (Norm) überschreiten, werden mit der Funktion `clip_grad_norm()` auf diese Norm beschnitten:

```
# Im Training-Loop, nach loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```

Typische Werte für `max_norm` sind 1.0 bis 10.0. Gradient Clipping ist bei Encoder-Decoder-Modellen fast immer empfehlenswert.

Ein Modell für automatische Übersetzung trainieren

Wir wollen nun ein Modell trainieren, das von English nach Deutsch übersetzt. Für maschinelle Übersetzung verwenden wir hier als Beispiel den Multi30k-Datensatz:

- **Aufgabe**: Englisch → Deutsch Übersetzung
- **Größe**: ca. 29.000 Trainings-, 1.000 Validierungspaare
- **Quelle**: Bildbeschreibungen von Flickr
- **Typische Satzlänge**: 10-30 Tokens

Da die Beispiel aus Bildbeschreibungen bestehen sind die Text relativ kurz und haben kein allzu großes Vokabular. Das erleichtert das Lernen mit relativ kleinen Modellen, so dass wir immer noch mit der CPU trainieren können. Das Modell wird sicher keine komplizierten Zeitungsartikel übersetzen können, aber probieren Sie ruhig einige Beispiele nach dem Training!

Zunächst müssen wir den Datensatz laden und ein Vokabular aufbauen. Wir erstellen separate Vokabulare für Englisch und Deutsch, da die Sprachen unterschiedliche Wörter haben. Zusätzlich fügen wir spezielle Tokens hinzu: `<PAD>` für Padding (Index 0), `<UNK>` für unbekannte Wörter, `<SOS>` für den Sequenzanfang und `<EOS>` für das Sequenzende.

```
import torch
from torch.utils.data import Dataset, DataLoader
from collections import Counter
```

```python
# Spezielle Tokens
PAD_IDX = 0
UNK_IDX = 1
SOS_IDX = 2
EOS_IDX = 3

def build_vocab(sentences, min_freq=2):
    """Baut ein Vokabular aus einer Liste von Sätzen."""
    counter = Counter()
    for sentence in sentences:
        counter.update(sentence.lower().split())

    # Spezielle Tokens zuerst
    vocab = {"<PAD>": PAD_IDX, "<UNK>": UNK_IDX, "<SOS>": SOS_IDX, "<EOS>": EOS_IDX}

    # Nur Wörter mit mindestens min_freq Vorkommen
    for word, count in counter.items():
        if count >= min_freq:
            vocab[word] = len(vocab)

    return vocab

def tokenize(sentence, vocab):
    """Wandelt einen Satz in Token-Indizes um."""
    tokens = [SOS_IDX]  # Start-Token
    for word in sentence.lower().split():
        tokens.append(vocab.get(word, UNK_IDX))
    tokens.append(EOS_IDX)  # End-Token
    return tokens

class TranslationDataset(Dataset):
    def __init__(self, src_sentences, trg_sentences, src_vocab, trg_vocab):
        self.src_data = [tokenize(s, src_vocab) for s in src_sentences]
        self.trg_data = [tokenize(s, trg_vocab) for s in trg_sentences]

    def __len__(self):
        return len(self.src_data)

    def __getitem__(self, idx):
        return self.src_data[idx], self.trg_data[idx]

def collate_fn(batch):
    """Padding für variable Sequenzlängen im Batch."""
    src_batch, trg_batch = zip(*batch)

    # Auf maximale Länge padden
    src_lens = [len(s) for s in src_batch]
    trg_lens = [len(t) for t in trg_batch]

    src_padded = torch.zeros(len(batch), max(src_lens), dtype=torch.long)
```

```
    trg_padded = torch.zeros(len(batch), max(trg_lens), dtype=torch.long)

    for i, (src, trg) in enumerate(zip(src_batch, trg_batch)):
        src_padded[i, :len(src)] = torch.tensor(src)
        trg_padded[i, :len(trg)] = torch.tensor(trg)

    return src_padded, trg_padded

# Multi30k-Daten laden (vereinfachtes Beispiel)
# In der Praxis würden Sie torchtext oder datasets verwenden
train_en = [
    "a man in a blue shirt is standing on a ladder .",
    "a girl is playing with a dog .",
    "two people are walking in the park .",
    # ... weitere Sätze
]
train_de = [
    "ein mann in einem blauen hemd steht auf einer leiter .",
    "ein mädchen spielt mit einem hund .",
    "zwei personen gehen im park spazieren .",
    # ... weitere Sätze
]

# Vokabulare erstellen
en_vocab = build_vocab(train_en)
de_vocab = build_vocab(train_de)

print(f"Englisches Vokabular: {len(en_vocab)} Wörter")
print(f"Deutsches Vokabular: {len(de_vocab)} Wörter")

# Dataset und DataLoader erstellen
dataset = TranslationDataset(train_en, train_de, en_vocab, de_vocab)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True, collate_fn=collate_fn)
```

Die `collate_fn`-Funktion ist wichtig, da sie Sequenzen unterschiedlicher Länge in einem Batch zusammenfasst, indem sie kürzere Sequenzen mit Nullen (dem Padding-Index) auffüllt.

Das Encoder-Decoder-Modell trainieren

Hier ist nun der Trainingscode, sie erkennen die Trainingsschleife sicher aus den vorherigen Kapiteln:

```
loss_fn = nn.CrossEntropyLoss(ignore_index=0)  # Padding ignorieren
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

def train_epoch(model, dataloader, loss_fn, optimizer):
    model.train()
    total_loss = 0

    for src, trg in dataloader:
        optimizer.zero_grad()
```

```
        # Forward Pass mit Teacher Forcing
        output = model(src, trg, teacher_forcing_ratio=0.5)

        # Reshape für Loss-Berechnung
        # Ignoriere <SOS> im Ziel
        output = output[:, 1:, :].reshape(-1, output.size(-1))
        trg = trg[:, 1:].reshape(-1)

        loss = loss_fn(output, trg)
        loss.backward()

        # Gradient Clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        optimizer.step()
        total_loss += loss.item()

    return total_loss / len(dataloader)
```

Nun können wir das Modell initialisieren und die Trainingsschleife ausführen:

```
# Modell erstellen
encoder = Encoder(embedding, HIDDEN_SIZE)
decoder = Decoder(embedding, HIDDEN_SIZE)
model = Seq2Seq(encoder, decoder)

# Training durchführen
num_epochs = 10

for epoch in range(num_epochs):
    loss = train_epoch(model, dataloader, loss_fn, optimizer)
    print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss:.4f}")
```

Nach dem Training können wir das Modell zur Übersetzung verwenden. Beachten Sie, dass wir bei der Inferenz kein Teacher Forcing verwenden, da wir die Zielsequenz nicht kennen:

```
def translate(model, src_sentence, src_vocab, trg_vocab, max_len=50):
    """Übersetzt einen englischen Satz ins Deutsche."""
    model.eval()

    # Umgekehrtes Vokabular für die Ausgabe
    idx_to_word = {idx: word for word, idx in trg_vocab.items()}

    # Eingabe tokenisieren
    src_tokens = tokenize(src_sentence, src_vocab)
    src_tensor = torch.tensor(src_tokens).unsqueeze(0)

    with torch.no_grad():
        hidden, cell = model.encoder(src_tensor)

        # Mit <SOS> starten
```

```
        decoder_input = torch.tensor([[SOS_IDX]])
        output_tokens = []

        for _ in range(max_len):
            output, hidden, cell = model.decoder(decoder_input, hidden, cell)
            predicted_idx = output.argmax(dim=-1).item()

            if predicted_idx == EOS_IDX:
                break

            output_tokens.append(idx_to_word.get(predicted_idx, "<UNK>"))
            decoder_input = torch.tensor([[predicted_idx]])

    return " ".join(output_tokens)

# Beispiel-Übersetzung
test_sentence = "a man is walking in the park ."
translation = translate(model, test_sentence, en_vocab, de_vocab)
print(f"Englisch: {test_sentence}")
print(f"Deutsch: {translation}")
```

Mit diesem Setup können Sie ein einfaches Übersetzungsmodell trainieren. Die Ergebnisse werden jedoch begrenzt sein, weil der Information Bottleneck bei längeren Sätzen zum Problem wird. Dafür lernen wir nun den Attention-Mechanismus kennen.

4.5 Attention-Mechanismus: Grundlagen

Das Problem, das Attention löst

Erinnern Sie sich an den Information Bottleneck: Der Encoder komprimiert die gesamte Eingabe in einen Vektor fester Größe. Bei einem Satz wie “The red car that I bought yesterday is parked outside” muss der Decoder für das deutsche Wort “rote” wissen, dass im Englischen “red” vorkam. Aber “red” wurde viele Wörter früher verarbeitet, die Information könnte im festen Kontext-Vektor verloren gegangen sein.

Die Frage ist: Warum alles komprimieren, wenn der Decoder für verschiedene Ausgabewörter verschiedene Teile der Eingabe braucht?

Die Attention-Idee

Die zentrale Idee von *Attention* (Aufmerksamkeit) ist einfach: Anstatt nur den finalen Encoder-State zu verwenden, geben wir dem Decoder Zugriff auf *alle* Encoder-States.

Für jedes Ausgabewort “schaut” der Decoder auf alle Eingabepositionen und entscheidet, welche relevant sind:

- Beim Übersetzen von “rote” → hohe Aufmerksamkeit auf “red”
- Beim Übersetzen von “Auto” → hohe Aufmerksamkeit auf “car”
- Bei anderen Wörtern → Aufmerksamkeit verteilt sich je nach Kontext

Das Ergebnis ist ein *dynamischer Kontext-Vektor* für jeden Decoder-Schritt: Eine gewichtete Summe aller Encoder-States, wobei die Gewichte die Relevanz jeder Position angeben.

Query, Key, Value verstehen

Attention verwendet drei Konzepte, die aus der Datenbankterminologie stammen:

Query (Q): Was suche ich? Das ist der aktuelle Decoder-Hidden-State. Er repräsentiert, welche Information der Decoder gerade braucht.

Key (K): Was biete ich an? Das sind alle Encoder-Hidden-States. Jede Position "bewirbt" sich mit ihrem Key.

Value (V): Was liefere ich? Ebenfalls die Encoder-Hidden-States. Wenn eine Position "ausgewählt" wird, wird ihr Value verwendet.

In der Praxis sind Key und Value oft identisch (beide die Encoder-Outputs), aber das Konzept der Trennung ermöglicht flexiblere Architekturen.

Der Ablauf lässt sich am besten an einem Übersetzungsbeispiel verstehen. Nehmen wir an, der Decoder generiert gerade das deutsche Wort "Katze" und muss entscheiden, welche Teile des englischen Satzes "The cat sits on the mat" relevant sind.

1. **Vergleiche Query mit allen Keys**: Der aktuelle Decoder-State (Query) wird mit jedem Encoder-State (Key) verglichen. Das Skalarprodukt misst die Ähnlichkeit. In unserem Beispiel wird der Query "ich suche ein Tier-Wort" mit allen Keys verglichen. Der Key für "cat" hat eine hohe Ähnlichkeit (z.B. Score 4.2), während "the" und "on" niedrige Ähnlichkeit haben (z.B. Scores 0.5 und 0.3).
2. **Normalisiere die Vergleichsergebnisse zu Wahrscheinlichkeiten**: Die Softmax-Funktion wandelt die Scores in Wahrscheinlichkeiten um, die sich zu 1 addieren. Aus den Scores [0.5, 4.2, 0.8, 0.3, 0.4, 1.1] für "The cat sits on the mat" werden nach Softmax etwa [0.02, 0.75, 0.03, 0.02, 0.02, 0.04]. Das Wort "cat" erhält 75% der Aufmerksamkeit.
3. **Berechne gewichtete Summe der Values**: Die Values (Encoder-States) werden mit den Wahrscheinlichkeiten gewichtet und aufsummiert. Der resultierende Kontext-Vektor enthält hauptsächlich die Information von "cat", gemischt mit kleinen Anteilen der anderen Wörter. Dieser Vektor hilft dem Decoder, "Katze" korrekt zu generieren.

Scaled Dot-Product Attention

Die häufigste Attention-Variante ist *Scaled Dot-Product Attention*:

```
Attention(Q, K, V) = softmax(Q · K^T / √d_k) · V
```

Schritt für Schritt:

1. **Dot Product**: Berechne `Q · K^T`. Das Skalarprodukt misst die Ähnlichkeit zwischen Query und jedem Key. Hohe Werte bedeuten hohe Ähnlichkeit.
2. **Skalierung**: Teile durch `√d_k` (Wurzel der Key-Dimension). Ohne Skalierung würden die Dot-Products bei großen Dimensionen sehr groß werden.

3. **Softmax**: Normalisiere zu Wahrscheinlichkeiten, die sich zu 1 addieren. Das sind die Attention-Gewichte.
4. **Gewichtete Summe**: Multipliziere Gewichte mit Values und summiere. Das ist der Kontext-Vektor.

Hier ist eine Implementierung:

```
import math

import torch
import torch.nn.functional as F


def scaled_dot_product_attention(query, key, value):
    d_k = query.size(-1)

    # Schritt 1 & 2: Scores berechnen und skalieren
    scores = torch.matmul(query, key.transpose(-2, -1))
    scores = scores / math.sqrt(d_k)

    # Schritt 3: Softmax für Attention-Gewichte
    attention_weights = F.softmax(scores, dim=-1)

    # Schritt 4: Gewichtete Summe der Values
    context = torch.matmul(attention_weights, value)

    return context, attention_weights


# Beispiel
batch_size = 2
src_len = 10
hidden_size = 64

# Decoder Hidden State ist der Query
query = torch.randn(batch_size, 1, hidden_size)

# Encoder Outputs sind Keys und Values
encoder_outputs = torch.randn(batch_size, src_len, hidden_size)

# Attention berechnen
context, weights = scaled_dot_product_attention(
    query=query, key=encoder_outputs, value=encoder_outputs
)

print(f"Context Shape: {context.shape}")
print(f"Attention Weights Shape: {weights.shape}")
print(f"Weights summieren zu 1: {weights.sum(dim=-1)}")
```

Die Ausgabe ist dann:

```
Context Shape: torch.Size([2, 1, 64])
Attention Weights Shape: torch.Size([2, 1, 10])
```

```
Weights summieren zu 1: tensor([[1.0000],
        [1.0000]])
```

Warum durch √d_k skalieren?

Die Skalierung ist wichtig für stabiles Training. Das Problem: Dot-Products wachsen mit der Dimension. Bei zwei zufälligen Vektoren der Dimension 256 ist der erwartete Dot-Product etwa 256-mal größer als bei Dimension 1.

Große Werte vor dem Softmax führen zu extremen Ausgaben: Fast alle Gewichte sind nahe 0, eines ist nahe 1. Das macht die Gradienten sehr klein (Softmax-Sättigung), und das Modell lernt schlecht.

Die Skalierung durch √d_k normalisiert die Varianz der Scores, unabhängig von der Dimension:

```
d_k = 256 → Scale by √256 = 16
d_k = 64  → Scale by √64 = 8
```

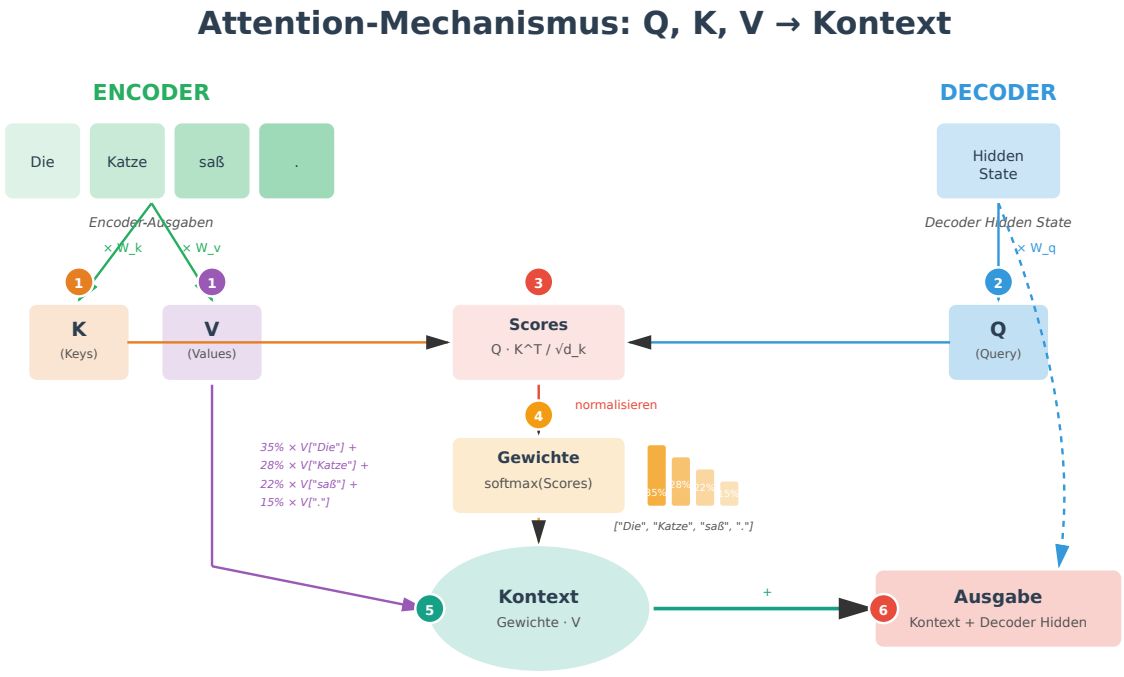

Abbildung 10: Scaled Dot-Product Attention: Query und Keys werden verglichen, durch Softmax normalisiert und dann mit den Values gewichtet.

Attention-Klasse mit lernbaren Projektionen

Nachdem wir die Grundlagen von Attention verstanden haben, implementieren wir nun eine vollständige Attention-Klasse. Der entscheidende Unterschied zur einfachen Funktion oben: Wir verwenden lernbare lineare Transformationen (Gewichtsmatrizen) für Query, Key und Value. Das ermöglicht dem Modell, selbst zu lernen, welche Aspekte der Eingabe für die Attention relevant sind. Die drei linearen Schichten `W_q`, `W_k` und `W_v` projizieren die Eingaben in einen "Attention-Raum", in dem die Ähnlichkeitsberechnungen stattfinden:

```python
class Attention(nn.Module):
    def __init__(self, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size

        # Lernbare Projektionen für Q, K, V
        self.W_q = nn.Linear(hidden_size, hidden_size)
        self.W_k = nn.Linear(hidden_size, hidden_size)
        self.W_v = nn.Linear(hidden_size, hidden_size)

        # Skalierungsfaktor vorberechnen
        self.scale = hidden_size ** 0.5

    def forward(self, query, key, value, mask=None):
        # Projektionen anwenden
        Q = self.W_q(query)
        K = self.W_k(key)
        V = self.W_v(value)

        # Attention Scores berechnen
        scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale

        # Optional: Maske anwenden (z.B. für Padding)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # Softmax und gewichtete Summe
        attention_weights = torch.softmax(scores, dim=-1)
        context = torch.matmul(attention_weights, V)

        return context, attention_weights
```

Die Maske ist wichtig, um Padding-Tokens zu ignorieren. Durch das Setzen der Scores auf `-1e9` vor dem Softmax werden diese Positionen effektiv ausgeblendet.

Attention-Gewichte visualisieren

Ein großer Vorteil von Attention: Die Gewichte sind interpretierbar. Sie zeigen, welche Eingabepositionen für jedes Ausgabewort relevant waren.

```
Source (EN): the  red  car  is  parked  outside
Target (DE): das rote Auto ist draußen geparkt

Attention für "rote":
  the:    0.05
  red:    0.80  ← Hohe Aufmerksamkeit
  car:    0.10
  is:     0.02
  parked: 0.02
  outside:0.01
```

Diese Visualisierung hilft beim Debugging und zeigt, ob das Modell sinnvolle Zusammenhänge gelernt hat.

Vorteile von Attention

Attention bringt mehrere wichtige Verbesserungen gegenüber dem reinen Encoder-Decoder-Ansatz. Der Information Bottleneck verschwindet, da alle Encoder-States für den Decoder verfügbar sind, nicht nur der letzte komprimierte Vektor. Das führt zu deutlich besserer Verarbeitung langer Sequenzen, weil der Decoder direkt auf frühe Eingabepositionen zugreifen kann, ohne dass diese Information erst durch viele RNN-Schritte "durchgereicht" werden muss.

Ein weiterer großer Vorteil ist die Interpretierbarkeit: Die Attention-Gewichte zeigen explizit, worauf das Modell bei jeder Vorhersage "achtet". Das erleichtert das Debugging und hilft zu verstehen, ob das Modell sinnvolle Zusammenhänge gelernt hat. In der Praxis führt Attention zu einer signifikanten Verbesserung von 10-20 BLEU-Punkten bei Übersetzungsaufgaben.

Schließlich ist Attention die konzeptuelle Grundlage für die Transformer-Architektur, die wir im nächsten Kapitel kennenlernen werden. Moderne Sprachmodelle wie GPT und BERT basieren vollständig auf Attention-Mechanismen.

4.6 Multi-Head Attention

In der Praxis wird selten nur ein einzelner Attention-Mechanismus verwendet. Stattdessen ist *Multi-Head Attention* der Standard in allen modernen Architekturen, von Seq2Seq-Modellen bis hin zu Transformern wie GPT und BERT. Die Idee: Anstatt nur eine Attention-Berechnung durchzuführen, führen wir mehrere parallele Attention-Operationen aus, jede mit eigenen lernbaren Gewichten. Diese parallelen Operationen nennt man "Heads" (Köpfe), typischerweise verwendet man 4 bis 16 davon.

Warum mehrere Attention Heads?

Ein einzelner Attention-Mechanismus kann nur eine "Perspektive" auf die Eingabe haben. Aber verschiedene Ausgabepositionen brauchen unterschiedliche Arten von Information:

- **Syntaktische Beziehungen**: Subjekt-Verb-Übereinstimmung
- **Semantische Beziehungen**: Bedeutungszusammenhänge
- **Positionelle Beziehungen**: Wortstellung

Die Idee von *Multi-Head Attention*: Führe mehrere Attention-Operationen parallel aus, jede mit eigenen lernbaren Gewichten. Jeder "Head" kann sich auf einen anderen Aspekt der Eingabe spezialisieren.

Multi-Head Attention Konzept

Multi-Head Attention funktioniert in vier Schritten. Nehmen wir an, wir haben `h = 4` Heads und eine Hidden-Dimension von 256.

Schritt 1: Separate Projektionen für jeden Head. Jeder Head hat seine eigenen Gewichtsmatrizen `W_i^Q`, `W_i^K` und `W_i^V`. Diese transformieren die Eingabe in head-spezifische Query-, Key- und Value-Repräsentationen. Head 1 könnte lernen, auf syntaktische Beziehungen zu achten, Head 2 auf semantische, und so weiter.

Schritt 2: Attention pro Head berechnen. Jeder Head führt seine eigene Scaled Dot-Product Attention durch. Das Ergebnis ist ein Kontext-Vektor pro Head. Formal: `head_i = Attention(Q · W_i^Q, K · W_i^K, V · W_i^V)`. Bei 4 Heads erhalten wir 4 verschiedene Kontext-Vektoren, die jeweils unterschiedliche Aspekte der Eingabe betonen.

Schritt 3: Konkatenation. Die Outputs aller Heads werden zu einem Vektor zusammengefügt. Wenn jeder Head einen 256-dimensionalen Output hat und wir 4 Heads verwenden, ist das konkatenierte Ergebnis 1024-dimensional: `Concat(head_1, head_2, head_3, head_4)`.

Schritt 4: Finale Projektion. Eine lineare Schicht `W^O` projiziert den konkatenierten Vektor zurück auf die ursprüngliche Dimension (256). Diese Schicht lernt, die verschiedenen Perspektiven der Heads optimal zu kombinieren.

Die vollständige Formel lautet:

```
MultiHead(Q, K, V) = Concat(head_1, ..., head_h) · W^O
```

Multi-Head Attention Implementierung

Die folgende Implementierung verwendet unsere `Attention`-Klasse von oben als Baustein. Der Hauptunterschied: Anstatt einer einzelnen Attention-Berechnung erstellen wir eine `ModuleList` mit mehreren Attention-Instanzen. Jeder Head hat dadurch seine eigenen lernbaren Gewichte. Am Ende werden die Outputs aller Heads konkateniert und durch eine zusätzliche lineare Schicht `out_linear` auf die ursprüngliche Dimension projiziert:

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, hidden_size, num_heads):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_heads = num_heads

        # Erstelle num_heads Attention-Module
        self.heads = nn.ModuleList([
            Attention(hidden_size) for _ in range(num_heads)
        ])

        # Ausgabeprojektion zum Kombinieren der Heads
        self.out_linear = nn.Linear(hidden_size * num_heads, hidden_size)

    def forward(self, query, key, value, mask=None):
        # Alle Heads parallel ausführen
        contexts = []
        all_weights = []

        for head in self.heads:
            context, weights = head(query, key, value, mask)
            contexts.append(context)
            all_weights.append(weights)

        # Outputs konkatenieren
        combined = torch.cat(contexts, dim=-1)
```

```
        # Auf hidden_size projizieren
        output = self.out_linear(combined)

        return output, all_weights
```

Wide vs. Narrow Attention

Es gibt zwei Varianten von Multi-Head Attention:

Narrow Attention (Standard in Transformers): Jeder Head arbeitet auf einem Teil der Dimensionen. - Bei `hidden_size=512` und `num_heads=8` hat jeder Head 64 Dimensionen - Effizienter, jeder Head lernt in einem anderen "Unterraum"

Wide Attention (in unseren Beispielen): Jeder Head arbeitet auf allen Dimensionen. - Einfacher zu verstehen und zu implementieren - Mehr Parameter, aber oft gute Ergebnisse

Für Lernzwecke ist Wide Attention klarer. In Produktionssystemen wird meist Narrow Attention verwendet.

Seq2Seq mit Multi-Head Attention

Nun integrieren wir Multi-Head Attention in unseren Decoder. Der wesentliche Unterschied zum Decoder ohne Attention liegt in der Informationsquelle: Der ursprüngliche Decoder erhielt nur den finalen Encoder-State als Initialisierung und musste dann "blind" generieren. Der Decoder mit Attention hingegen kann bei *jedem* Generierungsschritt auf alle Encoder-Outputs zugreifen und einen passenden Kontext-Vektor berechnen.

Die Implementierung verwendet Wide Attention, bei der jeder Head auf allen Dimensionen arbeitet. Das ist einfacher zu verstehen, führt aber zu mehr Parametern. In der LSTM-Eingabe konkatenieren wir das Embedding des aktuellen Tokens mit dem Kontext-Vektor, sodass das LSTM beide Informationsquellen nutzen kann:

```
class DecoderWithAttention(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size, num_heads):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.attention = MultiHeadAttention(hidden_size, num_heads)

        # LSTM-Eingabe: Embedding + Context
        self.lstm = nn.LSTM(
            embed_dim + hidden_size, hidden_size, batch_first=True
        )
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden, cell, encoder_outputs, src_mask=None):
        embedded = self.embedding(x)

        # Query ist der aktuelle Hidden State
        query = hidden[-1:].transpose(0, 1)

        # Attention berechnen
```

```
        context, attention_weights = self.attention(
            query, encoder_outputs, encoder_outputs, src_mask
        )

        # Embedded und Context konkatenieren
        lstm_input = torch.cat([embedded, context], dim=-1)

        # LSTM durchlaufen
        output, (hidden, cell) = self.lstm(lstm_input, (hidden, cell))
        prediction = self.fc(output)

        return prediction, hidden, cell, attention_weights
```

Attention-Gewichte visualisieren

Ein großer Vorteil von Attention-Modellen ist ihre Interpretierbarkeit. Die Attention-Gewichte zeigen für jedes generierte Ausgabewort, welche Eingabewörter besonders relevant waren. Das hilft nicht nur beim Verständnis des Modellverhaltens, sondern auch beim Debugging: Wenn das Modell falsche Übersetzungen produziert, können wir anhand der Attention-Gewichte sehen, ob es auf die falschen Wörter "geschaut" hat. Die folgende Funktion erstellt eine Heatmap, bei der dunklere Farben höhere Attention-Gewichte anzeigen:

```
import matplotlib.pyplot as plt
import numpy as np


def visualize_attention(src_tokens, trg_tokens, attention_weights, title):
    # attention_weights: Liste von [src_len] Tensoren, einer pro Output-Token
    attention_matrix = torch.stack(attention_weights).numpy().T

    fig, ax = plt.subplots(figsize=(10, 8))

    im = ax.imshow(attention_matrix, cmap='Blues', aspect='auto')

    ax.set_xticks(np.arange(len(trg_tokens)))
    ax.set_yticks(np.arange(len(src_tokens)))
    ax.set_xticklabels(trg_tokens, rotation=45, ha='right')
    ax.set_yticklabels(src_tokens)

    ax.set_xlabel("Target (German)")
    ax.set_ylabel("Source (English)")
    ax.set_title(title)

    plt.colorbar(im, ax=ax, label="Attention Weight")
    plt.tight_layout()

    return fig
```

Typische Muster in Attention-Heatmaps:

- **Diagonale**: Ähnliche Wortstellung in beiden Sprachen
- **Off-Diagonal-Blöcke**: Umordnung von Phrasen

- **Verteilte Aufmerksamkeit**: Ein Ausgabewort hängt von mehreren Eingabewörtern ab

Vergleich: Mit und ohne Attention

Die Verbesserung durch Attention ist dramatisch:

Modell	BLEU Score (Multi30k)
Seq2Seq ohne Attention	~15
Seq2Seq mit Attention	~25-30
Transformer (nur Attention)	~35+

Attention ermöglicht nicht nur bessere Übersetzungen, sondern beschleunigt auch das Training erheblich, da weniger Epochen nötig sind, um gute Ergebnisse zu erzielen. Zudem generalisieren Attention-Modelle deutlich besser auf längere Sequenzen, die sie während des Trainings nicht gesehen haben. Und schließlich liefern die Attention-Gewichte interpretierbare Vorhersagen, sodass wir nachvollziehen können, wie das Modell zu seinen Entscheidungen kommt.

Zusammenfassung

In diesem Kapitel haben Sie fortgeschrittene Techniken für sequenzielle Daten kennengelernt:

Packed Sequences lösen das Padding-Problem: - Entfernen Padding-Tokens vor der RNN-Verarbeitung - Liefern saubere Hidden States ohne Padding-Einfluss - `pack_padded_sequence` und `pad_packed_sequence` sind die Schlüsselfunktionen

LSTM erweitert GRU mit einem separaten Cell State: - Drei Gates (Forget, Input, Output) kontrollieren den Informationsfluss - Der Cell State ermöglicht Gradientenfluss über lange Sequenzen - Besonders effektiv bei komplexen, langreichweitigen Abhängigkeiten

Encoder-Decoder-Architektur ermöglicht Sequence-to-Sequence-Aufgaben: - Encoder komprimiert die Eingabe in einen Kontext-Vektor - Decoder generiert die Ausgabe autoregressive - Teacher Forcing beschleunigt das Training erheblich - Der Information Bottleneck begrenzt die Performance bei langen Sequenzen

Attention-Mechanismus löst den Information Bottleneck: - Decoder hat Zugriff auf alle Encoder-States - Query-Key-Value-Konzept ermöglicht dynamische Gewichtung - Scaled Dot-Product Attention ist die Standardvariante - Attention-Gewichte sind interpretierbar

Multi-Head Attention ermöglicht mehrere “Perspektiven”: - Mehrere Attention-Operationen parallel - Jeder Head kann unterschiedliche Muster lernen - Grundlage für die Transformer-Architektur

Im nächsten Kapitel werden wir die Transformer-Architektur kennenlernen, die RNNs komplett durch Attention ersetzt und die Grundlage für GPT, BERT und andere moderne Sprachmodelle bildet.

Kapitel 5: Die Transformer-Architektur

Im vorherigen Kapitel haben Sie den Attention-Mechanismus kennengelernt, der das Problem des Information Bottleneck in Encoder-Decoder-Modellen löst. Attention ermöglicht es dem Decoder, auf alle Positionen der Eingabe zuzugreifen, anstatt sich auf einen komprimierten Vektor verlassen zu müssen. In diesem Kapitel gehen wir den nächsten revolutionären Schritt: Wir ersetzen RNNs vollständig durch Attention.

Die Transformer-Architektur, vorgestellt 2017 im Paper "Attention Is All You Need", hat die Verarbeitung natürlicher Sprache grundlegend verändert. Transformer sind die Grundlage für alle modernen Sprachmodelle wie GPT, BERT und Claude. Sie werden verstehen, warum diese Architektur so erfolgreich ist und wie Sie sie selbst implementieren können.

5.1 Self-Attention und Positional Encoding

Was ist Self-Attention?

In Kapitel 4 haben wir Attention im Kontext von Encoder-Decoder-Modellen kennengelernt: Der Decoder (Query) "schaut" auf den Encoder (Keys/Values), um relevante Information zu finden. Aber Attention ist viel vielseitiger einsetzbar.

Bei *Self-Attention* schaut eine Sequenz auf sich selbst. Query, Key und Value kommen alle aus derselben Eingabesequenz. Jede Position kann auf jede andere Position "aufmerksam" sein, einschließlich auf sich selbst. Wir brauchen keine RNN-Zelle wie GRU oder LSTM mehr. Das vereinfacht auch die Berechnungen.

Betrachten wir den Satz "The cat sat on the mat":

- "cat" kann auf "sat" schauen (was macht die Katze?)
- "sat" kann auf "cat" schauen (wer sitzt?)
- "on" kann auf "mat" schauen (worauf wird gesessen?)

Im Encoder-Decoder-Attention aus Kapitel 4 kamen die Keys und Values vom Encoder (der Eingabesequenz) und die Query vom Decoder (der Ausgabesequenz). Bei Self-Attention gibt es diese Trennung nicht. Eine einzelne Sequenz liefert alle drei Komponenten. Dadurch kann das Modell Beziehungen *innerhalb* einer Sequenz lernen, ohne sie erst durch ein RNN sequenziell verarbeiten zu müssen.

Hier ist eine Implementierung von Self-Attention:

```
import math
```

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

torch.manual_seed(42)


class SelfAttention(nn.Module):
    def __init__(self, dimension_model):
        super().__init__()
        # Lineare Schichten projizieren Eingabe zu Q, K, V
        self.query = nn.Linear(dimension_model, dimension_model)
        self.key = nn.Linear(dimension_model, dimension_model)
        self.value = nn.Linear(dimension_model, dimension_model)

    def forward(self, x):
        # x: [batch, seq_len, dimension_model]
        Q = self.query(x)
        K = self.key(x)
        V = self.value(x)

        # Scaled Dot-Product Attention (aus Kapitel 4)
        d_k = Q.size(-1)
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)
        attention_weights = F.softmax(scores, dim=-1)
        output = torch.matmul(attention_weights, V)

        return output, attention_weights
```

Der entscheidende Punkt: Die gleiche Eingabe `x` wird durch drei verschiedene lineare Schichten geschickt, um Query, Key und Value zu erzeugen. Danach folgt die bekannte Scaled Dot-Product Attention aus Kapitel 4.

Testen wir die Implementierung:

```python
batch_size = 2
seq_len = 6
dimension_model = 8

self_attn = SelfAttention(dimension_model)
x = torch.randn(batch_size, seq_len, dimension_model)

output, weights = self_attn(x)
print(f"Input shape: {x.shape}")
print(f"Output shape: {output.shape}")
print(f"Attention weights: {weights.shape}")
```

```
Input shape: torch.Size([2, 6, 8])
Output shape: torch.Size([2, 6, 8])
Attention weights: torch.Size([2, 6, 6])
```

Die Attention-Gewichte haben die Form `[batch, seq_len, seq_len]`. Für jede der 6 Positionen gibt es 6 Gewichte, die angeben, wie stark diese Position auf jede andere Position "achtet".

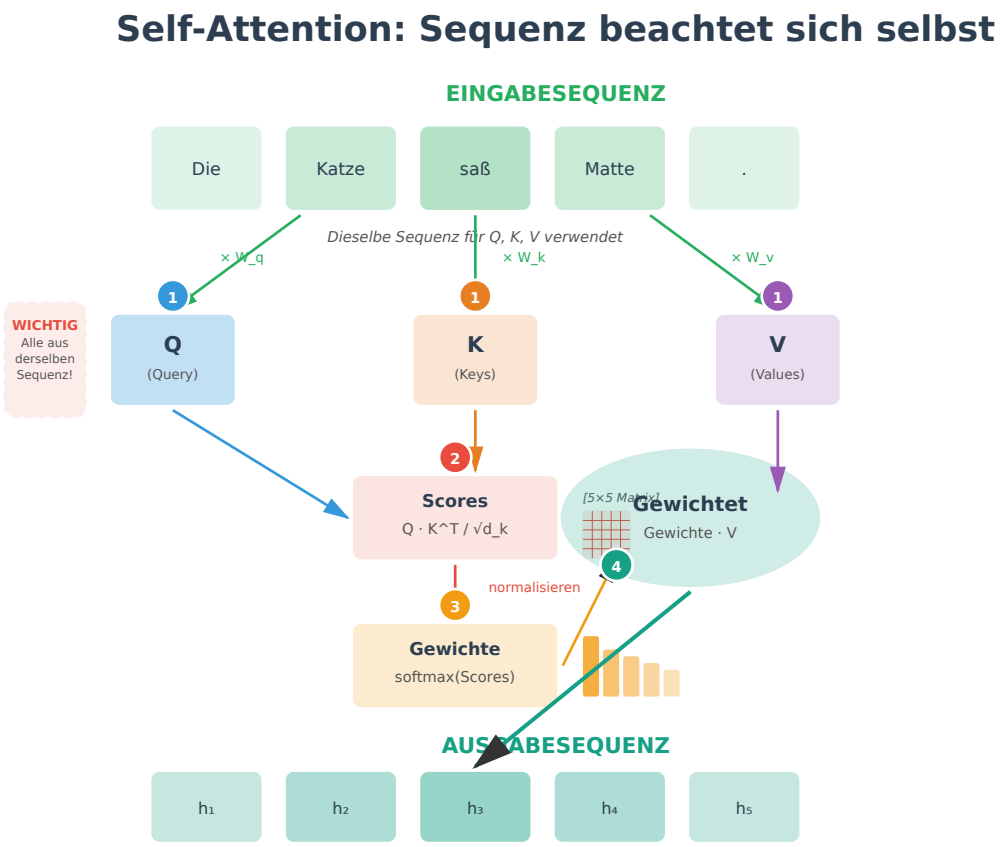

Abbildung 11: Self-Attention

Warum Self-Attention?

Self-Attention hat gegenüber RNNs mehrere entscheidende Vorteile.

Parallele Verarbeitung: Ein RNN muss Position für Position sequenziell verarbeiten. Position 5 kann erst berechnet werden, nachdem Positionen 1-4 fertig sind. Bei Self-Attention werden alle Positionen gleichzeitig verarbeitet. Das ermöglicht massive Parallelisierung auf GPUs.

Globaler Kontext sofort verfügbar: Bei einem RNN muss Information vom Anfang der Sequenz durch alle Zwischenpositionen "durchgereicht" werden, bevor sie am Ende ankommt. Bei Self-Attention sieht jede Position sofort die gesamte Sequenz.

Keine Distanzabhängigkeit: Im RNN ist die Verbindung zwischen weit entfernten Positionen durch viele Zwischenschritte geschwächt (das Vanishing-Gradient-Problem). Bei Self-Attention ist die Verbindung zwischen erstem und letztem Wort genauso direkt wie zwischen benachbarten Wörtern.

Flexible Beziehungen: Self-Attention kann beliebige Muster lernen, welche Wörter zusammengehören. Es muss keine feste Reihenfolge oder Struktur annehmen.

Das Positions-Problem

Self-Attention hat allerdings ein fundamentales Problem: Es hat kein Konzept von Position oder Reihenfolge. Die Attention-Operation ist *permutationsinvariant*, das bedeutet, wenn wir die Eingabepositionen vertauschen, ändern sich nur die entsprechenden Ausgabepositionen, aber die Berechnung selbst ist identisch.

Betrachten wir zwei Sätze: - "The cat sat" - "sat cat The"

Für die reine Attention-Berechnung sind diese identisch! Die gleichen Tokens erzeugen die gleichen Q, K, V Vektoren, nur in anderer Reihenfolge. Aber die Bedeutung ist völlig unterschiedlich.

In natürlicher Sprache ist Wortstellung aber entscheidend: - "Dog bites man" (Hund beißt Mann) - "Man bites dog" (Mann beißt Hund)

Gleiche Wörter, völlig andere Bedeutung. Wir müssen dem Modell irgendwie mitteilen, wo sich jedes Wort in der Sequenz befindet.

Positional Encoding mit Sinus und Cosinus

Um das Problem zu lösen addieren wir zu jedem Token-Embedding einen *Positionsvektor*, der die Position in der Sequenz kodiert. Das nennt man *Positional Encoding*.

Die originale Transformer-Architektur verwendet Sinus- und Cosinus-Funktionen mit unterschiedlichen Frequenzen:

Der Encoding-Tensor hat dieselbe Größe wie die Embedding-Dimension (`dimension_model` im Code unten). Jede Komponente dieses Tensors (Index 0, 1, 2, ..., dimension_model-1) bekommt einen eigenen Wert. Wir nennen diese Komponenten hier "Dimensionen":

- Für gerade Dimensionen (Index 0, 2, 4, ...): `sin(pos / 10000^(2i/dimension_model))`
- Für ungerade Dimensionen (Index 1, 3, 5, ...): `cos(pos / 10000^(2i/dimension_model))`

Dabei ist `pos` die Position im Satz (0, 1, 2, ...) und `i` die Dimension im Encoding-Vektor. Weiter unten gehen wir darauf ein, warum wir hier Sinus und Cosinus verwende. Hier zunächst eine Implementierung:

```
import math

import torch
import torch.nn as nn


class PositionalEncoding(nn.Module):
    def __init__(self, dimension_model, max_len=1000):
        super().__init__()

        # Positional Encoding Matrix erstellen
        pe = torch.zeros(max_len, dimension_model)

        # Positionsindizes: [0, 1, 2, ..., max_len-1]
        position = torch.arange(0, max_len).unsqueeze(1).float()

        # Divisionsterm für Frequenzen berechnen
        div_term = torch.exp(
            torch.arange(0, dimension_model, 2).float() * -(math.log(10000.0) / dime
        )

        # Sinus auf gerade Indizes, Cosinus auf ungerade Indices
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        # Batch-Dimension hinzufügen und als Buffer registrieren
        pe = pe.unsqueeze(0)  # [1, max_len, dimension_model]
        self.register_buffer("pe", pe)

    def forward(self, x):
        # x: [batch, seq_len, dimension_model]
        # Positional Encoding zur Eingabe addieren
        return x + self.pe[:, : x.size(1), :]
```

Gehen wir den Code Schritt für Schritt durch:

1. **Matrix erstellen**: Wir erstellen eine leere Matrix `pe` mit den Dimensionen `[max_len, dimension_model]`, also für jede mögliche Position einen Vektor.
2. **Positionsindizes**: Mit `torch.arange(0, max_len)` erzeugen wir die Positionen 0, 1, 2, ..., max_len-1. Das `unsqueeze(1)` fügt eine Dimension hinzu, damit wir später elementweise multiplizieren können.
3. **Frequenz-Divisor**: Der `div_term` berechnet die verschiedenen Frequenzen. Die Formel `exp(-log(10000) * 2i/d)` ist mathematisch äquivalent zu `1/10000^(2i/d)`. Niedrige Dimensionen bekommen hohe Frequenzen (schnelle Oszillation), hohe Dimensionen niedrige Frequenzen (langsame Oszillation).
4. **Sinus und Cosinus**: Mit der Slice-Notation `[::2]` bzw. `[1::2]` füllen wir die

geraden Indizes mit Sinus-Werten und die ungeraden mit Cosinus-Werten.

5. **Buffer registrieren**: `register_buffer` speichert den Tensor mit dem Modell, aber markiert ihn als nicht-trainierbar. Das Positional Encoding wird als "Buffer" registriert, das heißt, es wird mit dem Modell gespeichert, aber nicht trainiert. Es ist eine feste, vorberechnete Matrix.

Testen wir das Positional Encoding:

```
dimension_model = 8
pos_enc = PositionalEncoding(dimension_model, max_len=50)

batch_size = 2
seq_len = 10
embeddings = torch.randn(batch_size, seq_len, dimension_model)

output = pos_enc(embeddings)

print(f"Input shape: {embeddings.shape}")
print(f"Output shape: {output.shape}")
print(f"Positional Encoding shape: {pos_enc.pe.shape}")
```

Die Ausgabe ist dann:

```
Input shape: torch.Size([2, 10, 8])
Output shape: torch.Size([2, 10, 8])
Positional Encoding shape: torch.Size([1, 50, 8])
```

Die Form `[1, 50, 8]` des Positional Encodings bedeutet: Wir haben vorberechnete Encodings für bis zu 50 Positionen (weil wir `max_len=50` beim Erstellen angegeben haben), wobei jedes Encoding ein 8-dimensionaler Vektor ist (entsprechend `dimension_model=8`). Die führende 1 ist eine Batch-Dimension, die Broadcasting ermöglicht: Dasselbe Encoding wird für alle Elemente im Batch verwendet.

Warum Sinus- und Cosinus-Funktionen?

Die Wahl von Sinus und Cosinus ist nicht willkürlich. Sie haben mehrere nützliche Eigenschaften.

Unterschiedliche Frequenzen: Jede Dimension hat eine andere Wellenlänge. Die ersten Dimensionen haben sehr hohe Frequenzen (kurze Wellenlängen), die letzten sehr niedrige (lange Wellenlängen). Das ermöglicht dem Modell, sowohl feine Positionsunterschiede (benachbarte Wörter) als auch grobe (weit entfernte Wörter) zu erkennen.

Eindeutige Muster: Jede Position bekommt ein einzigartiges Encoding. Die Kombination aus Sinus und Cosinus über alle Dimensionen erzeugt für jede Position ein unterscheidbares Muster.

Lernbare relative Positionen: Mathematisch lässt sich zeigen, dass sich `PositinalEncoding(pos+k)` als lineare Funktion von `PositinalEncoding(pos)` ausdrücken lässt. Das bedeutet, das Modell kann lernen, "Abstand" zwischen Positionen zu verstehen, ohne die absolute Position kennen zu müssen.

Extrapolation: Da die Funktionen periodisch sind, kann das Modell auch mit Sequenzen umgehen, die länger sind als die während des Trainings gesehenen. Bei

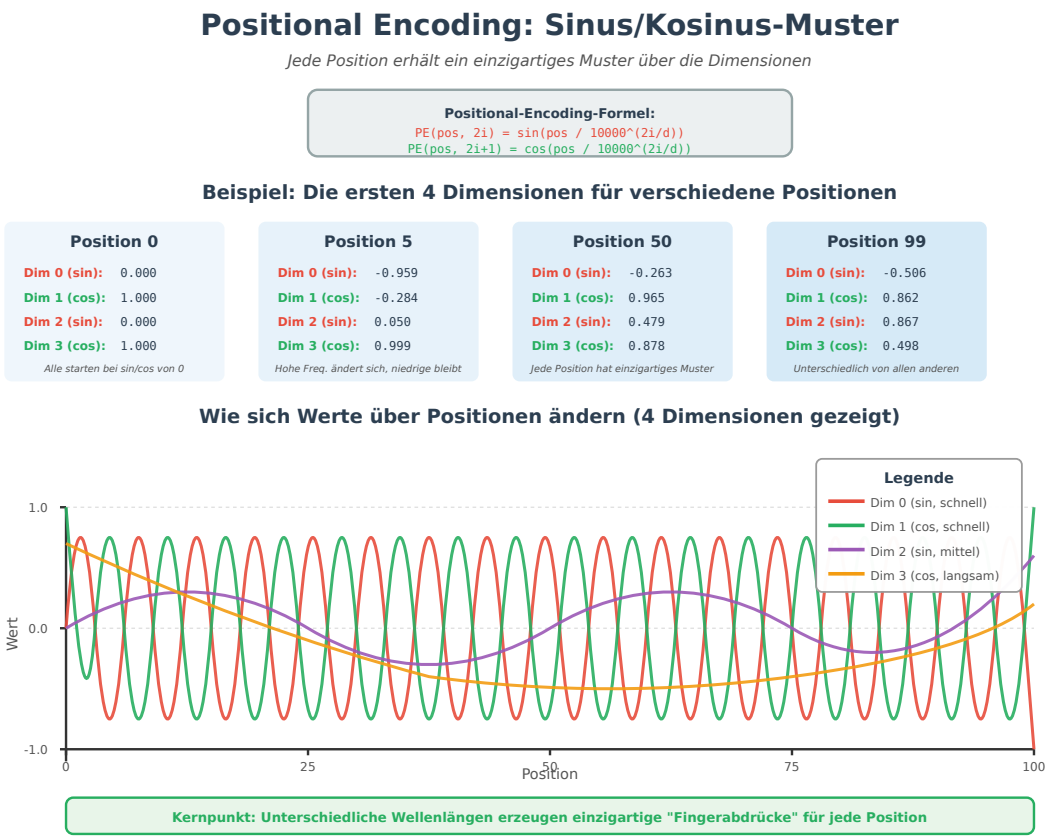

Abbildung 12: Positional Encoding

trainierten Position-Embeddings wäre das nicht möglich.

Keine Parameter: Das Encoding ist deterministisch und hat keine lernbaren Parameter. Das vereinfacht das Training.

Neuere Transformer-Varianten verwenden oft andere Positional Encodings, wie RoPE (Rotary Position Embedding) oder trainierbare Positionsembeddings. Das Grundprinzip bleibt aber gleich: Dem Modell muss mitgeteilt werden, wo sich jedes Token befindet.

5.2 Der Transformer-Encoder

Multi-Head Self-Attention

Im vorherigen Kapitel haben wir Multi-Head Attention kennengelernt: Mehrere parallele Attention-Operationen, die unterschiedliche Aspekte der Eingabe erfassen können. Im Transformer verwenden wir Multi-Head *Self*-Attention, also Multi-Head Attention, bei der Query, Key und Value alle aus derselben Eingabe stammen.

Hier ist eine Implementierung von Multi-Head Attention, die im Transformer verwendet wird. Im Gegensatz zur Wide-Attention-Variante aus Kapitel 4 teilen wir hier die Dimensionen auf die Heads auf (Narrow Attention):

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dimension_model, num_heads, dropout=0.1):
        super().__init__()
```

```
        self.dimension_model = dimension_model
        self.num_heads = num_heads
        self.head_dim = dimension_model // num_heads

        # Lineare Projektionen für Q, K, V
        self.W_q = nn.Linear(dimension_model, dimension_model)
        self.W_k = nn.Linear(dimension_model, dimension_model)
        self.W_v = nn.Linear(dimension_model, dimension_model)

        # Ausgabeprojektion
        self.W_o = nn.Linear(dimension_model, dimension_model)

        self.dropout = nn.Dropout(dropout)

    def split_heads(self, x):
        """Letzte Dimension in (num_heads, head_dim) aufteilen"""
        # x: [batch, seq_len, dimension_model]
        batch_size, seq_len, dimension_model = x.size()
        x = x.view(batch_size, seq_len, self.num_heads, self.head_dim)
        # Output: [batch, num_heads, seq_len, head_dim]
        return x.transpose(1, 2)

    def combine_heads(self, x):
        """Heads wieder zu dimension_model kombinieren"""
        # x: [batch, num_heads, seq_len, head_dim]
        batch_size, num_heads, seq_len, head_dim = x.size()
        # Output: [batch, seq_len, dimension_model]
        x = x.transpose(1, 2)
        return x.contiguous().view(batch_size, seq_len, self.dimension_model)

    def forward(self, q, k, v, mask=None):
        # Lineare Projektionen
        Q = self.W_q(q)
        K = self.W_k(k)
        V = self.W_v(v)

        # In mehrere Heads aufteilen
        Q = self.split_heads(Q)  # [batch, num_heads, seq_len, head_dim]
        K = self.split_heads(K)
        V = self.split_heads(V)

        # Scaled Dot-Product Attention
        d_k = Q.size(-1)
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)

        # Maske anwenden falls vorhanden
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float("-inf"))

        attention_weights = F.softmax(scores, dim=-1)
```

```
        attention_weights = self.dropout(attention_weights)

        # Attention auf Values anwenden
        attended_values = torch.matmul(attention_weights, V)

        # Heads kombinieren
        output = self.combine_heads(attended_values)

        # Finale lineare Projektion
        output = self.W_o(output)

        return output, attention_weights
```

Die `split_heads`-Methode ist zentral: Sie nimmt einen Tensor der Form `[batch, seq_len, dimension_model]` und formt ihn zu `[batch, num_heads, seq_len, head_dim]` um. Bei `dimension_model=64` und `num_heads=8` hat jeder Head 8 Dimensionen (`head_dim = 64 // 8 = 8`).

Wir können die Klasse `MultiHeadAttention` nun wie folgt verwenden:

```
dimension_model = 64
num_heads = 8
batch_size = 2
seq_len = 10

mha = MultiHeadAttention(dimension_model, num_heads)
x = torch.randn(batch_size, seq_len, dimension_model)

output, weights = mha(x, x, x)  # Self-Attention: q=k=v=x

print(f"Input shape: {x.shape}")
print(f"Output shape: {output.shape}")
print(f"Attention weights shape: {weights.shape}")
print(f"Number of heads: {num_heads}")
print(f"Dimensions per head: {dimension_model // num_heads}")
```

```
Input shape: torch.Size([2, 10, 64])
Output shape: torch.Size([2, 10, 64])
Attention weights shape: torch.Size([2, 8, 10, 10])
Number of heads: 8
Dimensions per head: 8
```

Die Output-Form `[2, 10, 64]` entspricht der Eingabe: 2 Sequenzen im Batch, jeweils 10 Positionen, und jede Position hat 64 Dimensionen. Die Attention-Gewichte haben die Form `[2, 8, 10, 10]`: Für jeden der 8 Attention-Heads gibt es eine 10×10-Matrix, die angibt, wie stark jede der 10 Positionen auf jede andere Position "achtet". Jeder Head kann dabei unterschiedliche Beziehungsmuster lernen.

Residual Connections

Tiefe neuronale Netze leiden unter dem Problem verschwindender Gradienten: Bei vielen Schichten werden die Gradienten beim Backpropagation immer kleiner, bis sie praktisch verschwinden. Transformer haben typischerweise 6-24 Encoder-Layer, das Training wäre ohne Gegenmaßnahmen sehr schwierig. Wir hatten das

Problem schon in Kapitel 3 und 4 kennen gelernt. In Kapitel 3 haben die Gates z.B. in GRU weiter geholfen, wir haben nun aber keine RNN-Zellen mehr.

Residual Connections (auch Skip Connections genannt) lösen dieses Problem elegant. Die Idee, bekannt aus ResNet für Computer Vision: Wir addieren die Eingabe einer Schicht direkt zu ihrer Ausgabe. Im Prinzip ein sehr einfaches Gate.

```
output = x + SubLayer(x)
```

Anstatt nur das Ergebnis der Attention zu verwenden, addieren wir die ursprüngliche Eingabe hinzu. Der Gradient kann nun auf zwei Wegen fließen: durch die Attention-Berechnung *und* direkt durch die Addition. Selbst wenn der Pfad durch die Attention kleine Gradienten hat, fließt der Gradient durch die Identitätsverbindung ungehindert.

Layer Normalization

Neben Residual Connections verwendet der Transformer *Layer Normalization*, um das Training zu stabilisieren. Im Gegensatz zu Batch Normalization, die wir in Kapitel 3 für CNNs verwendet haben, normalisiert Layer Normalization nicht über den Batch, sondern über die Features (Dimensionen) jedes einzelnen Elements.

Betrachten wir ein Beispiel mit einem Batch von 2 Sequenzen, jede mit 3 Positionen und 4 Features:

```
Batch:
  Sequenz 1: [[1, 2, 3, 4],   # Position 0
              [5, 6, 7, 8],   # Position 1
              [9, 10, 11, 12]]  # Position 2
  Sequenz 2: [[13, 14, 15, 16],
              [17, 18, 19, 20],
              [21, 22, 23, 24]]
```

Batch Normalization berechnet Mittelwert und Standardabweichung *über den Batch und die Sequenzpositionen*, aber separat für jedes Feature. Für Feature 0 (erste Spalte) wäre das der Mittelwert von [1, 5, 9, 13, 17, 21] = 11.

Layer Normalization berechnet Mittelwert und Standardabweichung *über die Features*, aber separat für jede Position. Für Position 0 von Sequenz 1 wäre das der Mittelwert von [1, 2, 3, 4] = 2.5.

Die Formel für Layer Normalization lautet:

```
LayerNorm(x) = γ * (x - mean) / std + β
```

Dabei sind γ (gamma) und β (beta) lernbare Parameter, die es dem Modell ermöglichen, die Normalisierung bei Bedarf rückgängig zu machen.

Warum Layer Normalization statt Batch Normalization? Batch Normalization funktioniert schlecht bei sequenziellen Daten mit variabler Länge, da Statistiken über den Batch weniger aussagekräftig sind. Layer Normalization normalisiert jede Sequenzposition unabhängig und ist daher besser für Transformer geeignet.

In der ursprünglichen Transformer-Architektur wird Layer Normalization *nach* jeder Sublayer angewendet (“Post-LN”). Modernere Implementierungen verwenden oft “Pre-LN”, wo die Normalisierung *vor* der Sublayer stattfindet:

```
# Post-LN (original):
x = LayerNorm(x + SubLayer(x))

# Pre-LN (moderner, stabiler):
x = x + SubLayer(LayerNorm(x))
```

Pre-LN führt oft zu stabileren Gradienten und besserer Konvergenz, besonders bei tiefen Modellen.

Position-Wise Feed-Forward Networks

Nach der Self-Attention folgt ein *Feed-Forward Network* (FFN), das auf jede Position unabhängig angewendet wird. Es besteht aus zwei linearen Schichten mit einer ReLU-Aktivierung dazwischen:

$$FFN(x) = \max(0, x \cdot W_1 + b_1) \cdot W_2 + b_2$$

Das FFN expandiert die Dimensionalität und komprimiert sie wieder. Bei `dimension_model=512` ist `dimension_ff` typischerweise `2048` (Faktor 4). Diese Expansion ermöglicht dem Netzwerk, komplexere Transformationen zu lernen.

```
feed_forward = nn.Sequential(
    nn.Linear(dimension_model, dimension_ff),    # 512 → 2048
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(dimension_ff, dimension_model),    # 2048 → 512
)
```

Das es sich um einfach lineare Abbildungen handelt wird das FFN *positionsweise* angewendet, das heißt, jede Position wird unabhängig von den anderen verarbeitet. Die Interaktion zwischen Positionen findet nur in der Self-Attention statt.

Der komplette Transformer Encoder Layer

Wir verwenden zwei separate Layer-Normalization-Schichten (`norm1` und `norm2`), weil jede Sublayer (Self-Attention und Feed-Forward) ihre eigene Normalisierung braucht. Jede Normalisierung hat ihre eigenen lernbaren Parameter (γ und β), die auf die spezifischen Aktivierungsmuster der jeweiligen Sublayer abgestimmt werden können.

Nun können wir alle Komponenten zu einem Encoder Layer zusammenfügen:

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, dimension_model, num_heads, dimension_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(dimension_model, num_heads, dropout)
        self.feed_forward = nn.Sequential(
            nn.Linear(dimension_model, dimension_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(dimension_ff, dimension_model),
        )
        self.norm1 = nn.LayerNorm(dimension_model)
        self.norm2 = nn.LayerNorm(dimension_model)
        self.dropout = nn.Dropout(dropout)
```

```
    def forward(self, x, mask=None):
        # Pre-LN Self-Attention mit Residual Connection
        attn_output, _ = self.self_attn(
            self.norm1(x), self.norm1(x), self.norm1(x), mask=mask
        )
        x = x + self.dropout(attn_output)

        # Pre-LN Feed-Forward mit Residual Connection
        ff_output = self.feed_forward(self.norm2(x))
        x = x + self.dropout(ff_output)

        return x
```

Die Architektur eines Encoder Layers ist also:

1. **Layer Norm** auf der Eingabe
2. **Multi-Head Self-Attention**
3. **Residual Connection** (Eingabe + Attention-Output)
4. **Layer Norm** auf dem Zwischenergebnis
5. **Feed-Forward Network**
6. **Residual Connection** (Zwischenergebnis + FFN-Output)

Der vollständige Transformer Encoder

Ein vollständiger Transformer Encoder besteht aus mehreren gestapelten Encoder Layers, zusammen mit Embedding und Positional Encoding. Wir normalisieren auch ein weiteres Mal nach allen Encoder Layers:

```
class SimpleTransformerEncoder(nn.Module):
    def __init__(self, vocab_size, dimension_model=128, num_heads=8,
                 num_layers=2, dimension_ff=512):
        super().__init__()

        # Embedding + Positional Encoding
        self.embedding = nn.Embedding(vocab_size, dimension_model)
        self.pos_encoding = PositionalEncoding(dimension_model)

        # Stapel von Encoder Layers
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(dimension_model, num_heads, dimension_ff)
            for _ in range(num_layers)
        ])

        # Finale Layer Normalization
        self.norm = nn.LayerNorm(dimension_model)

    def forward(self, x):
        # x: [batch, seq_len] - Token-Indizes

        # Embedding und Positional Encoding hinzufügen
        x = self.embedding(x)  # [batch, seq_len, dimension_model]
        x = self.pos_encoding(x)
```

```
        # Durch alle Encoder Layers
        for layer in self.layers:
            x = layer(x)

        # Finale Normalisierung
        x = self.norm(x)

        return x
```

Testen wir den Encoder:

```
vocab_size = 1000
encoder = SimpleTransformerEncoder(
    vocab_size, dimension_model=128, num_heads=8, num_layers=2
)

batch_size = 2
seq_len = 15
tokens = torch.randint(0, vocab_size, (batch_size, seq_len))

output = encoder(tokens)
print(f"Input tokens shape: {tokens.shape}")
print(f"Encoder output shape: {output.shape}")
print(f"Total parameters: {sum(p.numel() for p in encoder.parameters()):,}")
```

```
Input tokens shape: torch.Size([2, 15])
Encoder output shape: torch.Size([2, 15, 128])
Total parameters: 395,520
```

Der Encoder transformiert Token-Indizes in *kontextualisierte Repräsentationen*. Jede Position hat nun Information über alle anderen Positionen, gewichtet nach Relevanz.

5.3 Der Transformer-Decoder

Decoder als Encoder-Variante

Der Transformer-Decoder ist dem Encoder sehr ähnlich, hat aber zwei wichtige Unterschiede:

1. **Causal Masking**: Der Decoder darf nur vergangene Positionen sehen, nicht zukünftige
2. **Cross-Attention**: Der Decoder kann auf die Encoder-Outputs zugreifen. Dies geschieht über eine zusätzliche Attention-Schicht, bei der die Query vom Decoder kommt, aber Keys und Values vom Encoder stammen. So kann der Decoder bei jedem Generierungsschritt "nachschauen", welche Teile der Eingabe relevant sind.

Im Encoder-Decoder-Szenario (wie maschinelle Übersetzung) generiert der Decoder die Ausgabesequenz Token für Token, wobei er sowohl seine eigene bisherige Ausgabe als auch die komplette Eingabe vom Encoder berücksichtigt.

Causal Masking verstehen

Beim Training eines Sprachmodells oder Decoders gibt es ein wichtiges Problem: Wenn wir den Satz “Ich liebe Katzen” als Zielsequenz haben, darf das Modell beim Vorhersagen von “liebe” nur “Ich” sehen, nicht “Katzen”. Sonst würde es schummeln und die Antwort direkt ablesen.

Causal Masking (auch autoregressive Masking genannt) löst dieses Problem, indem es zukünftige Positionen in der Attention “ausblendet”. Die Attention-Scores für zukünftige Positionen werden auf negative Unendlichkeit gesetzt, sodass sie nach dem Softmax zu null werden.

```
def create_causal_mask(size):
    # Erstelle untere Dreiecksmatrix
    # 1 = erlaubt zu attenden, 0 = maskiert
    mask = torch.tril(torch.ones(size, size))
    return mask
```

Für eine Sequenz der Länge 5 sieht die Maske so aus:

```
causal_mask = create_causal_mask(5)
print("Causal Mask:")
print(causal_mask.int())
```

Die Ausgabe ist dann:

```
Causal Mask:
tensor([[1, 0, 0, 0, 0],
        [1, 1, 0, 0, 0],
        [1, 1, 1, 0, 0],
        [1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1]])
```

Die Interpretation der Zeilen: - Position 0 kann nur Position 0 sehen - Position 1 kann Positionen 0-1 sehen - Position 2 kann Positionen 0-2 sehen - Position 4 kann alle Positionen 0-4 sehen

Die Nullen werden in der Attention zu `-inf` gesetzt, sodass `softmax(-inf) = 0`. Die zukünftigen Positionen haben effektiv kein Attention-Gewicht.

Warum Causal Masking kritisch ist

Ohne Causal Masking würde das Modell beim Training “schummeln”. Es könnte einfach die Antwort aus der Zukunft ablesen, anstatt zu lernen, sie vorherzusagen. Das Ergebnis wäre ein Modell, das im Training perfekt abschneidet, aber bei der Inferenz (wenn es die Zukunft nicht kennt) völlig versagt.

Mit Causal Masking lernt das Modell, das nächste Wort nur aus dem bisherigen Kontext vorherzusagen. Das ist genau die Fähigkeit, die es bei der Textgenerierung braucht.

Wir verwenden die Maske auch bei der Inferenz. Obwohl es dann keine “Zukunft” gibt (wir generieren Token für Token), müssen wir trotzdem alle Positionen bei jedem Schritt neu berechnen. Die Maske stellt sicher, dass das konsistent zum Training geschieht.

Cross-Attention im Decoder

Neben der maskierten Self-Attention hat jeder Decoder Layer eine zusätzliche *Cross-Attention*-Schicht. Hier kommt die Query vom Decoder, aber Keys und Values kommen vom Encoder.

Das ist die gleiche Encoder-Decoder-Attention, die wir in Kapitel 4 kennengelernt haben, nur jetzt mit Multi-Head Attention:

- **Query**: Aktueller Zustand des Decoders ("Was suche ich?")
- **Key, Value**: Encoder-Outputs ("Was bietet die Eingabe?")

Die Cross-Attention ermöglicht es dem Decoder, bei jedem Generierungsschritt auf die relevanten Teile der Eingabe zu "schauen".

Der komplette Transformer Decoder Layer

Die Decoder-Layer-Implementierung unterscheidet sich in wenigen Punkten vom Encoder. Wir verwenden dieselbe `MultiHeadAttention`-Klasse, aber zweimal: einmal für Self-Attention (`self_attn`) und einmal für Cross-Attention (`cross_attn`). Entsprechend haben wir drei Layer-Normalization-Schichten (`norm1`, `norm2`, `norm3`) statt zwei, da jede der drei Sublayer ihre eigene Normalisierung braucht.

```
class TransformerDecoderLayer(nn.Module):
    def __init__(self, dimension_model, num_heads, dimension_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(dimension_model, num_heads, dropout)
        self.cross_attn = MultiHeadAttention(dimension_model, num_heads, dropout)
        self.feed_forward = nn.Sequential(
            nn.Linear(dimension_model, dimension_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(dimension_ff, dimension_model),
        )
        self.norm1 = nn.LayerNorm(dimension_model)
        self.norm2 = nn.LayerNorm(dimension_model)
        self.norm3 = nn.LayerNorm(dimension_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, encoder_output, src_mask=None, tgt_mask=None):
        # Self-Attention auf Zielsequenz mit Causal Mask
        attn_output, _ = self.self_attn(
            self.norm1(x), self.norm1(x), self.norm1(x), mask=tgt_mask
        )
        x = x + self.dropout(attn_output)

        # Cross-Attention zum Encoder Output
        attn_output, _ = self.cross_attn(
            self.norm2(x), encoder_output, encoder_output, mask=src_mask
        )
        x = x + self.dropout(attn_output)

        # Feed-Forward
        ff_output = self.feed_forward(self.norm3(x))
```

```
        x = x + self.dropout(ff_output)

        return x
```

Jeder Decoder Layer hat also drei Sublayer: 1. **Masked Self-Attention**: Der Decoder schaut auf seine eigene bisherige Ausgabe 2. **Cross-Attention**: Der Decoder schaut auf die Encoder-Outputs 3. **Feed-Forward Network**: Positionsweise Transformation

Der vollständige Transformer Decoder

Der vollständige Decoder kombiniert mehrere Decoder-Layer mit Embedding, Positional Encoding und einem finalen Klassifikations-Kopf. Die Causal Mask wird in der `forward`-Methode erstellt und an alle Layer weitergegeben. Die finale lineare Schicht (`fc`) projiziert die Decoder-Ausgabe auf die Vokabulargröße, um Wahrscheinlichkeiten für jedes mögliche nächste Token zu erzeugen.

```
class SimpleTransformerDecoder(nn.Module):
    def __init__(self, vocab_size, dimension_model=128, num_heads=8,
                 num_layers=2, dimension_ff=512):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, dimension_model)
        self.pos_encoding = PositionalEncoding(dimension_model)

        self.layers = nn.ModuleList([
            TransformerDecoderLayer(dimension_model, num_heads, dimension_ff)
            for _ in range(num_layers)
        ])

        self.fc = nn.Linear(dimension_model, vocab_size)
        self.norm = nn.LayerNorm(dimension_model)

    def forward(self, x, encoder_output):
        seq_len = x.size(1)

        x = self.embedding(x)
        x = self.pos_encoding(x)

        # Causal Mask erstellen
        mask = create_causal_mask(seq_len).unsqueeze(0).unsqueeze(0).to(x.device)

        # Durch Decoder Layers (mit Causal Mask)
        for layer in self.layers:
            x = layer(x, encoder_output, tgt_mask=mask)

        # Finale Normalisierung und Projektion zum Vokabular
        x = self.norm(x)
        logits = self.fc(x)  # [batch, seq_len, vocab_size]

        return logits
```

Der Decoder gibt Logits zurück, einen Vektor der Größe `vocab_size` für jede Posi-

tion. Diese können mit Softmax in Wahrscheinlichkeiten umgewandelt werden.

Teacher Forcing mit Transformer

Wie in Kapitel 4 verwenden wir auch beim Transformer-Training *Teacher Forcing*: Wir geben dem Decoder die echte Zielsequenz als Eingabe, nicht seine eigenen Vorhersagen.

Bei der Zielsequenz `[<SOS>, I, love, cats, <EOS>]`: - Eingabe an Decoder: `[<SOS>, I, love, cats]` (ohne letztes Token) - Erwartete Ausgabe: `[I, love, cats, <EOS>]` (ohne erstes Token, um eins verschoben)

Das Causal Masking stellt sicher, dass das Modell beim Vorhersagen von “love” nur `<SOS>` und “I” sieht, nicht “cats” oder `<EOS>`.

5.4 Kompletter Encoder-Decoder Transformer

Encoder und Decoder kombinieren

Nun fügen wir Encoder und Decoder zu einem vollständigen Seq2Seq-Transformer zusammen:

```
class Seq2SeqTransformer(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size,
                 dimension_model=256, num_heads=8, num_layers=4):
        super().__init__()

        self.encoder = SimpleTransformerEncoder(
            src_vocab_size, dimension_model, num_heads, num_layers
        )
        self.decoder = SimpleTransformerDecoder(
            tgt_vocab_size, dimension_model, num_heads, num_layers
        )

    def forward(self, src, tgt):
        # Quelle enkodieren
        encoder_output = self.encoder(src)  # [batch, src_len, dimension_model]

        # Ziel dekodieren (mit Encoder-Kontext)
        decoder_output = self.decoder(tgt, encoder_output)

        return decoder_output
```

Transformer vs. RNN Encoder-Decoder

Verglichen mit dem RNN-basierten Encoder-Decoder aus Kapitel 4 hat der Transformer mehrere Vorteile:

Aspekt	RNN Encoder-Decoder	Transformer
Verarbeitung	Sequenziell	Parallel

Aspekt	RNN Encoder-Decoder	Transformer
Langreichweitige Abhängigkeiten	Schwierig (Vanishing Gradients)	Direkt (Attention)
Trainingsgeschwindigkeit	Langsam	Schnell
Speicherbedarf	O(n)	$O(n^2)$ für Attention-Matrix

Der quadratische Speicherbedarf der Attention-Matrix ist der Hauptnachteil des Transformers. Bei einer Sequenz von 1000 Tokens benötigt die Attention-Matrix 1.000.000 Einträge. Bei sehr langen Dokumenten kann das problematisch werden. Die Optimierung des Attention-Mechanismus sowohl im Hinblick auf Geschwindigkeit als auch auf Speicherbedarf ist heutzutage eines der aktivsten Forschungsgebiete im Bereich Deep Learning.

Training für maschinelle Übersetzung

Hier ist ein vollständiges Trainingsbeispiel für einen Übersetzungs-Transformer. Wir verwenden den Multi30k-Datensatz (Englisch-Deutsch), den wir bereits in Kapitel 4 verwendet haben:

Der folgende Code verwendet dieselben Hilfsfunktionen und Datenverarbeitung wie in Kapitel 4. Die Datenaufbereitung (Tokenisierung, Vokabularerstellung, Dataset-Klasse) bleibt identisch, da der Transformer dieselbe Eingabestruktur erwartet. Der wesentliche Unterschied liegt in den Modellparametern und der Trainingsschleife.

Die Konfiguration enthält neue Transformer-spezifische Parameter:

```
# Neue Transformer-Parameter (zusätzlich zu Kapitel 4)
dimension_model = 256  # Embedding-Dimension
NUM_HEADS = 8          # Anzahl der Attention-Heads
NUM_LAYERS = 3         # Anzahl der Encoder/Decoder-Layer
```

Die Trainingsschleife für den Transformer:

```
# Modell erstellen
model = Seq2SeqTransformer(
    len(src_vocab),
    len(trg_vocab),
    dimension_model=dimension_model,
    num_heads=NUM_HEADS,
    num_layers=NUM_LAYERS,
).to(DEVICE)

print(f"Modell-Parameter: {sum(p.numel() for p in model.parameters()):,}")

# Training
loss_fn = nn.CrossEntropyLoss(ignore_index=0)  # Padding ignorieren
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

```python
def train_epoch(model, dataloader, loss_fn, optimizer, device):
    model.train()
    total_loss = 0

    for src, trg in dataloader:
        src, trg = src.to(device), trg.to(device)

        optimizer.zero_grad()

        # Forward Pass: alle außer letztem Token als Eingabe
        trg_input = trg[:, :-1]
        output = model(src, trg_input)

        # Loss: vergleiche mit allen außer erstem Token
        output = output.reshape(-1, output.size(-1))
        trg_output = trg[:, 1:].reshape(-1)

        loss = loss_fn(output, trg_output)
        loss.backward()

        # Gradient Clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        optimizer.step()
        total_loss += loss.item()

    return total_loss / len(dataloader)


# Training durchführen
for epoch in range(NUM_EPOCHS):
    train_loss = train_epoch(model, train_loader, loss_fn, optimizer, DEVICE)
    print(f"Epoch {epoch + 1}/{NUM_EPOCHS}: Loss: {train_loss:.4f}")
```

Nach dem Training können wir das Modell zur Übersetzung verwenden:

```python
def translate(model, src_text, src_vocab, trg_vocab, device, max_len=50):
    """Übersetzt einen Quellsatz in die Zielsprache"""
    model.eval()

    # Umgekehrtes Vokabular für die Ausgabe
    idx_to_trg = {idx: word for word, idx in trg_vocab.items()}

    # Tokenisieren und zu Indizes konvertieren
    tokens = tokenize(src_text)
    indices = [src_vocab.get(token, src_vocab["<UNK>"]) for token in tokens]

    # Padding
    if len(indices) > max_len:
        indices = indices[:max_len]
    indices = indices + [src_vocab["<PAD>"]] * (max_len - len(indices))
```

```
    src_tensor = torch.tensor([indices]).to(device)

    with torch.no_grad():
        # Enkodieren
        encoder_output = model.encoder(src_tensor)

        # Mit <SOS> starten
        tgt_indices = [trg_vocab["<SOS>"]]

        for _ in range(max_len):
            tgt_tensor = torch.tensor([tgt_indices]).to(device)

            # Dekodieren
            output = model.decoder(tgt_tensor, encoder_output)

            # Vorhersage für letztes Token
            pred_token = output[0, -1, :].argmax().item()

            # Stoppen bei <EOS>
            if pred_token == trg_vocab["<EOS>"]:
                break

            tgt_indices.append(pred_token)

        # Zu Wörtern konvertieren
        translation = [idx_to_trg[idx] for idx in tgt_indices[1:]]

    return " ".join(translation)


# Testübersetzungen
test_sentences = [
    "two young guys with shaggy hair look at their hands",
    "a brown and white dog is running through the snow",
]

print("\nTestübersetzungen:")
for src in test_sentences:
    translation = translate(model, src, src_vocab, trg_vocab, DEVICE)
    print(f"EN: {src}")
    print(f"DE: {translation}\n")
```

Vorteile und Trade-offs

Der Transformer bietet gegenüber RNN-basierten Modellen deutliche Vorteile. Das Training läuft erheblich schneller, da alle Positionen parallel verarbeitet werden können. Während ein RNN Position für Position sequenziell durcharbeiten muss, berechnet der Transformer die Attention für alle Positionen gleichzeitig. Das ermöglicht eine viel bessere Auslastung moderner GPUs.

Langreichweitige Abhängigkeiten werden besser modelliert, da jede Position direkt auf jede andere zugreifen kann. Bei RNNs muss Information durch alle Zwi-

schenpositionen fließen, was zu Vanishing Gradients führt. Im Transformer ist die Verbindung zwischen dem ersten und dem letzten Wort genauso direkt wie zwischen benachbarten Wörtern.

Die Architektur ist konzeptionell einfacher, da sie nur aus Attention-Schichten und Feed-Forward-Netzwerken besteht. Es gibt keine komplexen Gating-Mechanismen wie bei LSTMs oder GRUs. Diese Einfachheit hat dazu geführt, dass Transformer bei praktisch allen NLP-Benchmarks State-of-the-art-Ergebnisse liefern.

Allerdings gibt es auch Trade-offs. Der Speicherbedarf wächst quadratisch mit der Sequenzlänge, da die Attention-Matrix alle Paare von Positionen speichert. Bei einer Sequenz von 4096 Tokens benötigt diese Matrix bereits über 16 Millionen Einträge pro Head. Bei sehr langen Dokumenten kann das problematisch werden und erfordert spezielle Techniken wie Sparse Attention.

Anders als RNNs haben Transformer kein inhärentes Konzept von Reihenfolge. Die Attention-Operation selbst ist permutationsinvariant, weshalb wir explizit Positional Encodings hinzufügen müssen. Außerdem haben Transformer typischerweise mehr Parameter als vergleichbare RNNs, was mehr Trainingsdaten und Rechenleistung erfordert.

5.5 Transformer-Varianten: BERT, GPT, T5

Drei Haupt-Architekturen

Der ursprüngliche Transformer ist ein Encoder-Decoder-Modell für Sequenz-zu-Sequenz-Aufgaben. Aber die Transformer-Bausteine lassen sich auch anders kombinieren. Es haben sich drei Hauptvarianten etabliert:

Encoder-Only (BERT): Nur der Encoder-Teil, ohne Decoder. Verwendet bidirektionale Attention (jede Position sieht alle anderen). Der Encoder erzeugt für jede Position eine kontextualisierte Repräsentation, die die Information des gesamten Textes einfließen lässt. Für Klassifikation wird typischerweise die Repräsentation eines speziellen `[CLS]`-Tokens verwendet und durch einen Klassifikations-Kopf geleitet. Für Token-Level-Aufgaben wie Named Entity Recognition wird die Repräsentation jedes Tokens einzeln klassifiziert. Ein Decoder ist nicht nötig, da keine neue Sequenz generiert wird, sondern nur die Eingabe analysiert wird.

Decoder-Only (GPT): Nur der Decoder-Teil, ohne Encoder. Verwendet Causal Masking (jede Position sieht nur Vergangenheit). Der Decoder generiert Token für Token, wobei jedes neue Token auf Basis aller bisherigen Tokens vorhergesagt wird. Ein separater Encoder ist nicht nötig, da die Eingabe (der Prompt) einfach als Präfix behandelt wird. Das Modell "vervollständigt" den Prompt, indem es das wahrscheinlichste nächste Token vorhersagt. Diese Architektur ist optimal für Generierungs-Aufgaben wie Textgenerierung, Vervollständigung oder Chatbots.

Encoder-Decoder (T5): Die vollständige Architektur. Der Encoder ist bidirektional, der Decoder causal. Optimal für Sequenz-zu-Sequenz-Aufgaben wie Übersetzung, Zusammenfassung oder Frage-Antwort-Systeme.

BERT: Encoder-Only für Verstehen

BERT (Bidirectional Encoder Representations from Transformers) wurde 2018 von Google vorgestellt und revolutionierte NLP. Die Kernidee: Ein Encoder-Only-

Modell, das bidirektional trainiert wird.

BERT verwendet nur den Encoder-Teil des Transformers, typischerweise 12-24 Layer. Da es keinen Decoder gibt, gibt es auch kein Causal Masking. Jede Position kann alle anderen Positionen sehen. BERT wird mit einer cleveren Aufgabe trainiert: Zufällig werden 15% der Tokens "maskiert" (durch ein spezielles `[MASK]`-Token ersetzt), und das Modell muss sie vorhersagen:

```
Eingabe: "The [MASK] sat on the mat"
Ziel: Vorhersage von "cat" an der maskierten Position
```

Das ist anders als bei GPT, das das *nächste* Wort vorhersagt. BERT muss ein Wort aus seinem *gesamten Kontext* (links und rechts) vorhersagen.

Nach dem Pre-Training auf großen Textmengen wird BERT für spezifische Aufgaben "fine-getuned". Für Klassifikation fügt man einen Klassifikations-Kopf hinzu und trainiert auf gelabelten Daten. Das vorgelernte Sprachverständnis macht das Fine-Tuning effizient.

GPT: Decoder-Only für Generierung

GPT (Generative Pre-trained Transformer) ist die Architektur hinter ChatGPT und vielen anderen modernen Sprachmodellen. Im Gegensatz zu BERT verwendet GPT nur den Decoder-Teil mit Causal Masking.

Die Architektur von GPT ist überraschend einfach und unterscheidet sich von unserer Seq2Seq-Implementierung in einem wesentlichen Punkt: Es gibt keinen separaten Encoder und keine Cross-Attention. Stattdessen werden die `TransformerEncoderLayer` (die wir bereits implementiert haben) mit Causal Masking verwendet. Die Eingabe ist der Prompt, und das Modell generiert einfach eine Fortsetzung. Im Gegensatz zum Seq2Seq-Modell, das zwei getrennte Sequenzen verarbeitet (Eingabe und Ausgabe), behandelt GPT alles als eine einzige Sequenz:

```
class SimpleGPT(nn.Module):
    def __init__(self, vocab_size, dimension_model=256, num_heads=8,
                 num_layers=4, max_len=512):
        super().__init__()

        # Embeddings
        self.token_embedding = nn.Embedding(vocab_size, dimension_model)
        self.pos_encoding = PositionalEncoding(dimension_model, max_len)

        # Decoder Layers (mit Causal Masking)
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(dimension_model, num_heads, dimension_model * 4)
            for _ in range(num_layers)
        ])

        # Output Head
        self.norm = nn.LayerNorm(dimension_model)
        self.lm_head = nn.Linear(dimension_model, vocab_size)

    def forward(self, input_ids):
        batch_size, seq_len = input_ids.size()
```

```
        # Embeddings und Positional Encoding
        x = self.token_embedding(input_ids)
        x = self.pos_encoding(x)

        # Causal Mask erstellen
        causal_mask = create_causal_mask(seq_len).to(x.device)

        # Durch Decoder Layers
        for layer in self.layers:
            x = layer(x, mask=causal_mask)

        # Finale Normalisierung und Projektion zum Vokabular
        x = self.norm(x)
        logits = self.lm_head(x)

        return logits
```

Obwohl wir `TransformerEncoderLayer` verwenden, ist es durch das Causal Masking effektiv ein Decoder. Der entscheidende Unterschied zum vollständigen Transformer-Decoder: Es gibt keine Cross-Attention, weil es keinen Encoder gibt, auf den man "schauen" könnte.

Sprache abbilden als Aufgabe

GPT wird mit einer einfachen, aber mächtigen Aufgabe trainiert: *Language Modeling*, also das Vorhersagen des nächsten Wortes.

```
Eingabe: "The cat sat on"
Ziel:    "cat sat on the"
```

Für jede Position in der Eingabe soll das Modell das nächste Token vorhersagen. Da die Eingabe um eins verschoben das Ziel ergibt, braucht man nur rohen Text als Trainingsdaten, keine manuellen Labels.

Das klingt trivial, aber um diese Aufgabe gut zu lösen, muss das Modell erstaunlich viel lernen: - Grammatik und Syntax - Semantik und Bedeutung - Weltwissen und Fakten - Logisches Schließen

Was die Forschungsgemeinschaft überraschte: Diese scheinbar einfache Aufgabe der Textvorhersage führt zu Modellen, die eine erstaunliche Bandbreite von Fähigkeiten entwickeln. GPT-3 zeigte 2020, dass ein ausreichend großes Modell Aufgaben lösen kann, für die es nie explizit trainiert wurde. Gibt man dem Modell einige Beispiele im Prompt ("Few-Shot Learning"), kann es Übersetzungen anfertigen, Code schreiben, mathematische Probleme lösen oder Fragen beantworten. Bei ChatGPT stellte sich heraus, dass Nutzer das Modell für Aufgaben verwenden, die niemand vorhergesehen hatte: Bewerbungsschreiben, Gedichte im Stil bestimmter Autoren, Erklärungen komplexer Konzepte für verschiedene Zielgruppen, oder das Debuggen von Programmcode.

Die Qualität der Vorhersagen skaliert mit Modellgröße und Datenmenge, was zur Ära der Large Language Models geführt hat.

Text-Generierung mit GPT

Die Textgenerierung mit GPT funktioniert *autoregressive*: Das Modell generiert ein Token nach dem anderen, wobei jedes neue Token zur Eingabe für den nächsten Schritt wird.

```
def generate_text(model, start_tokens, max_length=50, temperature=1.0):
    model.eval()
    generated = start_tokens.clone()

    with torch.no_grad():
        for _ in range(max_length):
            # Vorhersagen für aktuelle Sequenz
            logits = model(generated)

            # Logits für letzte Position
            next_logits = logits[0, -1, :] / temperature

            # Aus Verteilung samplen
            probs = F.softmax(next_logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1)

            # An Sequenz anhängen
            generated = torch.cat([generated, next_token.unsqueeze(0)], dim=1)

    return generated
```

Der Prozess: 1. Starte mit einem Prompt (einige Start-Tokens) 2. Führe das Modell aus, um Logits für alle Positionen zu bekommen 3. Nimm die Logits der letzten Position und konvertiere zu Wahrscheinlichkeiten 4. Sample ein Token aus dieser Verteilung 5. Hänge das Token an die Sequenz an 6. Wiederhole ab Schritt 2

Temperatur-Sampling

Im Code oben sehen Sie `torch.multinomial(probs, num_samples=1)`. Diese Funktion *sampelt* ein Token aus der Wahrscheinlichkeitsverteilung. Das bedeutet: Anstatt immer das wahrscheinlichste Token zu nehmen (das wäre "greedy decoding"), wird zufällig ein Token gezogen, wobei wahrscheinlichere Tokens häufiger gewählt werden. Ein Token mit 60% Wahrscheinlichkeit wird in etwa 60% der Fälle gewählt, eines mit 10% in etwa 10% der Fälle. Das führt zu abwechslungsreicheren, weniger repetitiven Texten.

Der `temperature`-Parameter kontrolliert die "Kreativität" der Generierung. Mathematisch teilen wir die Logits durch die Temperatur, bevor wir Softmax anwenden. Bei einer Temperatur von 1.0 bleiben die Wahrscheinlichkeiten unverändert. Eine niedrigere Temperatur (z.B. 0.5) bedeutet, dass wir die Logits durch einen kleineren Wert teilen, was die Unterschiede zwischen ihnen vergrößert. Stellen Sie sich vor, die Logits wären [2.0, 1.0, 0.5]. Teilen wir durch 0.5, bekommen wir [4.0, 2.0, 1.0]. Nach Softmax dominiert das wahrscheinlichste Token noch stärker. Bei hoher Temperatur (z.B. 2.0) werden die Unterschiede kleiner: [1.0, 0.5, 0.25]. Die Wahrscheinlichkeitsverteilung wird "flacher", und auch weniger wahrscheinliche Tokens haben eine realistische Chance, gewählt zu werden:

Niedrige Temperatur (z.B. 0.5): Die Wahrscheinlichkeitsverteilung wird "ge-

schärft". Das wahrscheinlichste Token wird noch wahrscheinlicher, unwahrscheinliche werden noch unwahrscheinlicher. Das Ergebnis ist fokussierterer, deterministischerer Text.

Hohe Temperatur (z.B. 1.5): Die Verteilung wird "geglättet". Auch weniger wahrscheinliche Tokens haben eine Chance. Das Ergebnis ist diverserer, kreativerer, aber potenziell auch unsinnigerer Text.

Temperatur 1.0: Die Standardverteilung, wie sie das Modell gelernt hat.

```
# Vergleiche verschiedene Temperaturen
for temp in [0.5, 1.0, 1.5]:
    output = generate_text(gpt, start_prompt, max_length=10, temperature=temp)
    print(f"Temperature {temp}: {output.tolist()}")
```

In der Praxis verwendet man oft Werte zwischen 0.7 und 1.0. Zu niedrig führt zu repetitivem, langweiligem Text, zu hoch zu inkoherentem Unsinn.

GPT-Skalierung und moderne LLMs

Die Geschichte der GPT-Modelle zeigt eindrucksvoll, wie Skalierung die Fähigkeiten verbessert:

- **GPT-1** (2018): 117 Millionen Parameter
- **GPT-2** (2019): 1,5 Milliarden Parameter
- **GPT-3** (2020): 175 Milliarden Parameter
- **GPT-4** (2023): Geschätzt über 1 Billion Parameter

Die zentrale Erkenntnis: Mehr Parameter + mehr Trainingsdaten = bessere Leistung. Ab einer gewissen Größe entstehen sogar "emergente Fähigkeiten", also Fähigkeiten, die kleinere Modelle nicht haben, wie komplexes Reasoning, Few-Shot Learning oder das Befolgen von Instruktionen.

Moderne Large Language Models (LLMs) wie GPT-4, Claude oder Llama basieren alle auf der GPT-Architektur (Decoder-Only mit Causal Masking), unterscheiden sich aber in Trainingsdaten, Größe und zusätzlichen Trainingstechniken wie RLHF (mehr dazu weiter unten).

T5: Encoder-Decoder für alles

T5 (Text-to-Text Transfer Transformer) von Google verfolgt einen anderen Ansatz: Anstatt spezialisierte Architekturen für verschiedene Aufgaben zu verwenden, formuliert T5 *alle* Aufgaben als Text-zu-Text-Probleme.

Beispiele: - **Übersetzung**: "translate English to French: I love cats" → "J'aime les chats" - **Klassifikation**: "sentiment: This movie is great" → "positive" - **Zusammenfassung**: "summarize: [langer Text]" → "Zusammenfassung" - **Frage-Antwort**: "question: What is the capital? context: Paris is the capital of France." → "Paris"

T5 verwendet die vollständige Encoder-Decoder-Architektur. Der Encoder verarbeitet die Eingabe bidirektional, der Decoder generiert die Ausgabe autoregressiv. Durch die einheitliche Text-zu-Text-Schnittstelle kann dasselbe Modell für viele verschiedene Aufgaben verwendet werden. Wir haben ein solches (kleineres) Modell oben in diesem Kapitel implementiert.

5.6 Moderne Sprachverarbeitung und Foundation Models

In den bisherigen Abschnitten haben wir die Transformer-Architektur von Grund auf implementiert. In der Praxis baut man jedoch selten alles selbst. Stattdessen nutzt man vortrainierte Modelle und spezialisierte Werkzeuge. In diesem Abschnitt lernen Sie die wichtigsten Konzepte und Tools kennen, die moderne Sprachverarbeitung ausmachen: effiziente Tokenisierung, Foundation Models, Fine-Tuning und die Hugging Face Transformers-Bibliothek.

Moderne Tokenisierung

In den bisherigen Beispielen haben wir einfache Wort-Tokenisierung verwendet: Jedes Wort wird zu einem eigenen Token. Das hat Nachteile:

- **Riesiges Vokabular**: Natürliche Sprache hat Millionen von Wörtern. Gerade wenn wir riesige Dokumentensammlungen verwenden, wie z.B. einen Großteil des Internetinhalts, dann explodiert die Anzahl der Wörter.
- **Out-of-Vocabulary (OOV)**: Unbekannte Wörter können nicht verarbeitet werden. Wir haben bei großen Textmengen immer ein gewisse Zahl an unbekannten Wörtern.
- **Keine Morphologie**: “walk”, “walked”, “walking” sind komplett separate Tokens. Das muss zunächst kein Nachteil sein, weil das Modell die Bedeutung separat lernen kann. Allerdings wird die Anzahl der Wörter wiederum größer. Die Bedeutung von Morpheme wie “-ed” könnten mit anderen Ansätzen auch getrennt gelernt werden.

Auf der anderen Seite wäre Zeichen-Tokenisierung (jedes Zeichen ein Token) auch problematisch: - Sehr lange Sequenzen - Schwierig, Wortbedeutungen zu lernen

Die Lösung ist *Subword-Tokenisierung*: Ein Kompromiss zwischen Wort- und Zeichen-Ebene. Häufige Wörter bleiben ganze Tokens, seltene werden in kleinere Einheiten zerlegt.

Byte-Pair Encoding (BPE)

Byte-Pair Encoding ist die populärste Subword-Tokenisierung. Der Algorithmus ist erstaunlich einfach:

1. Starte mit einzelnen Zeichen als Vokabular
2. Finde das häufigste benachbarte Zeichenpaar
3. Füge dieses Paar als neues Token zum Vokabular hinzu
4. Wiederhole bis gewünschte Vokabulargröße erreicht

```
from collections import Counter

# Startsequenzen: Wörter aufgeteilt in Zeichen
vocab = {"l o w": 5, "l o w e r": 4, "n e w e s t": 3}


def get_most_frequent_pair(vocab):
    """Finde häufigstes benachbartes Token-Paar"""
    pairs = Counter()
    for word, freq in vocab.items():
```

```
        tokens = word.split()
        for i in range(len(tokens) - 1):
            pairs[(tokens[i], tokens[i + 1])] += freq
    return pairs.most_common(1)[0][0] if pairs else None


def merge_pair(vocab, pair):
    """Verschmelze ein Paar benachbarter Tokens zu einem Token"""
    new_vocab = {}
    old = " ".join(pair)
    new = "".join(pair)
    for word, count in vocab.items():
        new_vocab[word.replace(old, new)] = count
    return new_vocab


print("BPE Merges:")
for i in range(5):
    pair = get_most_frequent_pair(vocab)
    if not pair:
        break
    print(f"{i + 1}. '{pair[0]}' + '{pair[1]}' → '{pair[0] + pair[1]}'")
    vocab = merge_pair(vocab, pair)
```

Das Ergebnis: Häufige Wörter wie “low” werden zu einem Token, seltene wie “newest” werden zu “new” + “est” zerlegt. Das “-est”-Suffix kann dann auch für andere Wörter wiederverwendet werden.

GPT-2 und GPT-3 verwenden BPE mit etwa 50.000 Tokens. Das ermöglicht effiziente Verarbeitung bei gleichzeitiger Handhabung seltener Wörter.

Die Transformers Library von Hugging Face

Für praktische Anwendungen verwendet man selten eigene Implementierungen. Die *Transformers*-Bibliothek von Hugging Face ist der De-facto-Standard. Hier ein Beispiel, wie wir das GPT2-Modell laden und anwenden:

```
import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Vortrainiertes Modell und Tokenizer laden
model_name = "gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)

# Eingabetext tokenisieren
text = "The future of artificial intelligence is"
input_ids = tokenizer.encode(text, return_tensors="pt")
print(f"Input tokens: {input_ids}")

# Text generieren
model.eval()
with torch.no_grad():
```

```
    output = model.generate(
        input_ids,
        max_length=50,
        num_return_sequences=1,
        temperature=0.8,
        do_sample=True,
        pad_token_id=tokenizer.eos_token_id,
    )

# Ausgabe dekodieren
generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print(f"Generiert: {generated_text}")
```

Die Transformers-Bibliothek bietet: - Tausende vortrainierte Modelle (GPT-2, BERT, T5, Llama, ...) - Einheitliche API für Tokenisierung und Inferenz - Einfaches Fine-Tuning auf eigenen Daten - Unterstützung für PyTorch und TensorFlow

Es lohnt sich, die Hugging Face Model Hub-Seite (https://huggingface.co/models) zu erkunden. Dort finden Sie tausende vortrainierte Modelle für verschiedene Aufgaben und Sprachen, darunter auch viele deutschsprachige Modelle. Jedes Modell hat eine Dokumentationsseite mit Beispielcode, Leistungswerten und Lizenzinformationen. Sie können Modelle nach Aufgabe (Textklassifikation, Übersetzung, etc.), Sprache oder Größe filtern.

Foundation Models verstehen

Foundation Models (auch “Grundlagenmodelle”) sind große, auf riesigen Datenmengen vortrainierte Modelle, die als Basis für viele Downstream-Aufgaben dienen.

Die Idee ist zweistufig:

1. **Pre-Training**: Das Modell wird auf einer selbst-supervisierten Aufgabe trainiert (wie Language Modeling), für die keine manuellen Labels nötig sind. Das ermöglicht Training auf Billionen von Tokens aus dem Internet, Büchern und anderen Quellen.
2. **Adaptation**: Das vortrainierte Modell wird an spezifische Aufgaben angepasst, entweder durch Fine-Tuning, Prompting oder andere Techniken.

Bekannte Foundation Models: - **GPT-4, Claude, Gemini**: Decoder-Only, hauptsächlich für Textgenerierung - **BERT, RoBERTa**: Encoder-Only, hauptsächlich für Verstehensaufgaben - **T5, BART**: Encoder-Decoder, für Sequenz-zu-Sequenz-Aufgaben - **CLIP, DALL-E**: Multimodal (Text + Bild)

Was ist Fine-Tuning?

Fine-Tuning ist der Prozess, ein vortrainiertes Modell an eine spezifische Aufgabe anzupassen. Anstatt ein Modell von Grund auf zu trainieren, startet man mit den gelernten Gewichten und trainiert mit einer kleineren Lernrate auf aufgabenspezifischen Daten weiter.

Der typische Ablauf:

1. Lade ein vortrainiertes Modell (z.B. BERT)

2. Füge einen aufgabenspezifischen “Kopf” (Layer) hinzu (z.B. eine Klassifikationsschicht)
3. Trainiere auf gelabelten Daten der Zielaufgabe
4. Verwende eine niedrigere Lernrate als beim Pre-Training

Um das Training zu beschleunigen trainiert man dabei häufig nicht alle Schichten des Foundation Models, sondern nur den aufgabenspezifischen Kopf und eventuell die letzten Layer des Ursprungmodells. Wie viele Schichten hier Sinn machen probiert man am Besten anhand der Aufgabe und der Ausgabe während und nach dem Training.

Die Vorteile von Fine-Tuning sind erheblich. Sie benötigen deutlich weniger Trainingsdaten, da das Modell bereits ein tiefes Sprachverständnis aus dem Pre-Training mitbringt. Während das Pre-Training hunderte Millionen Beispiele erfordert, reichen für Fine-Tuning oft wenige tausend oder sogar nur einige hundert Beispiele. Das Training ist entsprechend schneller und dauert oft nur wenige Epochen statt Wochen. Die Ergebnisse sind typischerweise besser als bei einem von Grund auf trainierten Modell, weil das vortrainierte Wissen über Grammatik, Semantik und Weltwissen direkt genutzt wird.

Hier ein Vergleich von Fine-Tuning gegenüber dem Training eines neuen Modells von Anfang an:

Aspekt	Pre-Training	Fine-Tuning
Datenmenge	100+ Millionen Beispiele	1.000-100.000 Beispiele
Rechenaufwand	Wochen-Monate auf vielen GPUs	Stunden auf einer GPU
Initialisierung	Zufällig	Vortrainierte Gewichte
Lernrate	Höher (z.B. 1e-4)	Niedriger (z.B. 1e-5)

Für die meisten praktischen Anwendungen ist Fine-Tuning der richtige Ansatz. Pre-Training lohnt sich nur, wenn Sie Zugang zu enormen Ressourcen haben und keine existierenden Modelle Ihre Anforderungen erfüllen.

RLHF: Reinforcement Learning from Human Feedback

Modelle, die nur auf Textvorhersage trainiert wurden, haben ein Problem: Sie können zwar grammatikalisch korrekten Text produzieren, aber dieser ist nicht unbedingt hilfreich, ehrlich oder sicher. Ein Modell könnte technisch korrekt, aber unhöflich, irreführend oder sogar schädlich antworten.

RLHF (Reinforcement Learning from Human Feedback) adressiert dieses Problem, indem menschliche Präferenzen ins Training einbezogen werden. Die Grundidee: Menschen bewerten Modellantworten, und das Modell lernt, Antworten zu generieren, die Menschen bevorzugen.

RLHF war ein entscheidender Durchbruch für Systeme wie ChatGPT. Vor RLHF produzierten Sprachmodelle zwar flüssigen Text, aber dieser war oft nicht hilfreich. Das Modell konnte auf eine Frage mit einer anderen Frage antworten, Unsinn erfinden oder unhöflich sein. RLHF löste diese Probleme, indem es dem Modell beibrachte, was Menschen tatsächlich wollen: direkte, hilfreiche Antworten statt Textfortsetzungen im Stil der Trainingsdaten. Der Qualitätssprung von GPT-3 zu

ChatGPT war dramatisch. Obwohl die Basisarchitektur ähnlich war, machte RLHF das Modell zu einem nützlichen Assistenten statt einem reinen Textgenerator.

Der RLHF-Trainingsprozess

RLHF besteht aus mehreren Phasen:

Phase 1: Supervised Fine-Tuning (SFT)

Zuerst wird das vortrainierte Modell auf hochqualitativen Beispielen fine-getuned. Diese Beispiele werden oft von Menschen geschrieben und zeigen das gewünschte Verhalten: hilfreiche, korrekte, sichere Antworten.

Phase 2: Reward Model Training

Dann wird ein "Belohnungsmodell" trainiert, das menschliche Präferenzen vorhersagt:

1. Das SFT-Modell generiert mehrere Antworten für denselben Prompt
2. Menschen ranken die Antworten (A ist besser als B)
3. Ein Reward Model wird trainiert, diese Rankings vorherzusagen

Das Reward Model lernt zu bewerten, wie "gut" eine Antwort ist, ohne dass für jede mögliche Antwort ein menschliches Label nötig wäre.

Phase 3: RL Fine-Tuning

Schließlich wird das Sprachmodell mit Reinforcement Learning optimiert:

1. Das Modell generiert Antworten
2. Das Reward Model bewertet sie
3. Das Sprachmodell wird optimiert, um höhere Bewertungen zu bekommen

Die "Policy" ist das Sprachmodell, die "Reward" kommt vom Reward Model. Typischerweise wird der PPO-Algorithmus (Proximal Policy Optimization) verwendet.

Das Ergebnis: Modelle wie ChatGPT, Claude oder Gemini, die nicht nur Text generieren können, sondern auch versuchen, hilfreich, harmlos und ehrlich zu sein.

Zusammenfassung

In diesem Kapitel haben Sie die Transformer-Architektur kennengelernt, die Grundlage für alle modernen Sprachmodelle:

Self-Attention und Positional Encoding: - Bei Self-Attention schaut eine Sequenz auf sich selbst - Parallele Verarbeitung statt sequenzieller RNN-Berechnungen - Positional Encoding fügt Positionsinformation hinzu - Sinus/Cosinus-Funktionen ermöglichen Extrapolation zu längeren Sequenzen

Der Transformer-Encoder: - Multi-Head Self-Attention für mehrere "Perspektiven" - Residual Connections für Gradientenfluss - Layer Normalization für stabiles Training - Position-Wise Feed-Forward Networks für komplexere Transformationen

Der Transformer-Decoder: - Causal Masking verhindert "Schummeln" durch Blick in die Zukunft - Cross-Attention verbindet Decoder mit Encoder - Teacher Forcing während des Trainings

Transformer-Varianten: - **BERT** (Encoder-Only): Bidirektional, für Verstehensaufgaben - **GPT** (Decoder-Only): Causal Masking, für Generierung - **T5** (Encoder-Decoder): Text-zu-Text für alle Aufgaben

Moderne Sprachverarbeitung: - Subword-Tokenisierung (BPE) als Kompromiss zwischen Wort- und Zeichen-Ebene - Foundation Models als vortrainierte Basis - Fine-Tuning zur Anpassung an spezifische Aufgaben - RLHF für Alignment mit menschlichen Präferenzen

Die Transformer-Architektur hat die künstliche Intelligenz revolutioniert. Von Übersetzung über Chatbots bis hin zu Codegenerierung, die Prinzipien, die Sie in diesem Kapitel gelernt haben, bilden das Fundament der modernsten KI-Systeme.