Agenda

1. Grundlagen von Predictive Analytics und Machine Learning
2. Datenaufbereitung und -exploration
3. Auswahl und Entwicklung von Vorhersagemodellen mit PyTorch
4. Praktische Übungen zur Implementierung von Deep Learning-Modellen
5. Fehleranalyse und Modellverbesserung
6. Implementierung von Vorhersagemodellen in PyTorch
7. Fallstudien und Anwendungen in verschiedenen Branchen
8. Integration von Predictive Analytics und Deep Learning in Geschäftsprozesse
9. Abschlussdiskussion und Empfehlungen für die weitere Entwicklung

1. Grundlagen von Predictive Analytics und Machine Learning

Definition und Bedeutung von Predictive Analytics

- Vorhersage zukünftiger Ereignisse basierend auf historischen Daten
- Einsatzmöglichkeiten in verschiedenen Branchen
- Geschäftlicher Mehrwert durch datengetriebene Entscheidungen

Beispiel: Wie funktioniert das?

🎯 Beispiel 1: Predictive Maintenance

🛠️ Maschinen vorausschauend warten, bevor Fehler auftreten

Datenquellen:

- Sensordaten aus Maschinen
- Historische Fehleranalysen
- Temperaturen, Druck, Laufzeiten

Ergebnis:

> "Mit 85% Wahrscheinlichkeit fällt die Maschine in den nächsten 7 Tagen aus."
> 👉 Instandhaltung kann gezielt geplant werden!

Kundenabwanderung vorhersagen (Churn Prediction)

Problem: Ein Unternehmen verliert Kunden, aber warum?

🔍 **Lösung: Predictive Analytics analysiert Kundendaten:**

✅ Kaufhistorie

✅ Website-Interaktionen

✅ Service-Anfragen

📈 **Ergebnis:**

- Kunden mit hoher Abwanderungswahrscheinlichkeit identifizieren
- Gezielte Angebote & Rabatte, um Kunden zu halten!

Einsatzmöglichkeiten in verschiedenen Branchen

💡 Predictive Analytics wird in vielen Branchen eingesetzt:

Finanzwesen & Versicherung

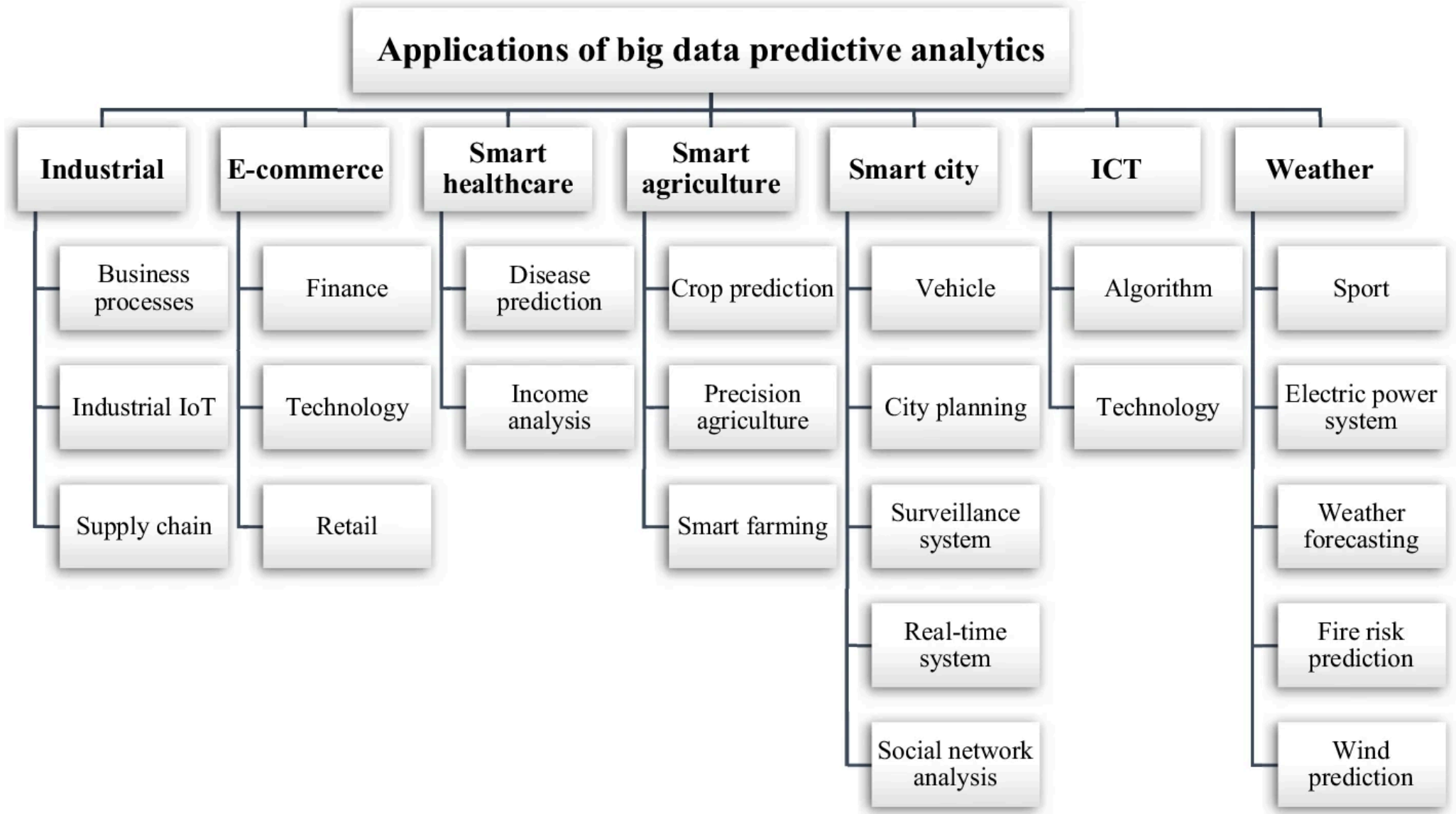
- Betrugserkennung (Fraud Detection)
- Markttrends & Aktienkursprognosen

Einzelhandel & Marketing

- Kundenverhalten vorhersagen
- Empfehlungssysteme (z. B. Amazon, Netflix)

Automobil & Fertigung

- Wartungsprognosen & Produktionsoptimierung
- Qualitätssicherung & Fehlerentdeckung


Applications of big data predictive analytics
Industrial
Business processes
Industrial IoT
Supply chain
E-commerce
Finance
Technology
Retail
Smart healthcare
Disease prediction
Income analysis
Smart agriculture
Crop prediction
Precision agriculture
Smart farming
Smart city
Vehicle
City planning
Surveillance system
Real-time system
Social network analysis
ICT
Algorithm
Technology
Weather
Sport
Electric power system
Weather forecasting
Fire risk prediction
Wind prediction

Herausforderungen

Vergangenheit kann die Zukunft nicht immer vorhersagen

- Die Verwendung historischer Daten zur Vorhersage der Zukunft setzt voraus, dass es bestimmte **konstante Bedingungen** oder **Steady-State-Bedingungen** in einem komplexen System gibt.
- Dies ist **fast immer falsch**, wenn das System Menschen involviert.
- **Beispiel**:
    - Finanzkrisen können nicht allein durch historische Daten vorhergesagt werden, da sich Marktbedingungen und menschliches Verhalten ständig ändern.

Das Problem der unbekannten Merkmale

- Bei der Datenakquise definiert der Benutzer zunächst die Variablen, für die Daten erhoben werden.
- Es besteht jedoch immer die Möglichkeit, dass **kritische Variablen** nicht berücksichtigt oder sogar definiert wurden.
- **Beispiel**:
    - In der Medizin können unbekannte genetische Faktoren oder Umweltbedingungen den Ausgang einer Behandlung beeinflussen, obwohl sie nicht in den ursprünglichen Daten enthalten waren.

Selbstzerstörung von Algorithmen

- Wenn ein Algorithmus zum akzeptierten Standard wird, kann er von Personen ausgenutzt werden, die den Algorithmus verstehen und ein Interesse daran haben, das Ergebnis zu manipulieren.
- Dies führt zu einer **Selbstzerstörung** des Algorithmus, da er nicht mehr zuverlässig ist.
- **Beispiel**:
  - **CDO-Ratings vor der Finanzkrise 2008**:
    - CDO-Händler manipulierten die Eingabevariablen, um AAA-Ratings für ihre Produkte zu erhalten, was zur Finanzkrise beitrug.

Verschiedene Arten von Vorhersagemodellen

Welche Modelle gibt es & wann werden sie verwendet?

- Supervised Learning (Überwachtes Lernen)
- Unsupervised Learning (Unüberwachtes Lernen)
- Time Series Forecasting (Zeitreihenanalyse)

Üblich: hybrides mathematisches Modell.

Überwachtes Lernen (Supervised Learning)

- Modell lernt aus beschrifteten Daten (`Eingabe → Ausgabe`)
- Ziel: **Vorhersage basierend auf historischen Daten**

Beispiele für überwachtes Lernen

Regression (kontinuierliche Werte)

- Vorhersage von Hauspreisen

Klassifikation (diskrete Klassen)

- Spam-Filter erkennt E-Mails als „Spam“ oder „Nicht-Spam“ ✉️

Anwendungsfälle:

- Medizinische Diagnosen
- Finanzmarkt-Prognosen
- Kundenbindung vorhersagen (Churn Prediction)

Beispiel: Spam-Erkennung mit Maschinellem Lernen

Wie erkennt ein ML-Algorithmus Spam?
Ein Machine-Learning-Modell kann helfen, unerwünschte E-Mails (Spam) zu klassifizieren, indem es Muster in den Nachrichten analysiert.

1️⃣ Datensammlung und Vorbereitung

- 🔹 Datenquellen: E-Mails mit Label „Spam“ oder „Nicht-Spam“
- 🔹 Merkmale (Features):
  - Bestimmte Wörter („Gewonnen“, „Gratis“, „Schnell Geld“)
  - Anzahl der Links
  - Verwendung von Großbuchstaben
  - Absender-Adresse
  - Länge und Struktur der E-Mail

2️⃣ Training eines Modells

Supervised Learning:

- 🔹 Das System wird mit vielen gelabelten E-Mails trainiert.
- 🔹 Ziel: Regeln lernen, die Spam von legitimen E-Mails unterscheiden.

Beispielhafte Vorgehensweise:

- Wortanalyse: Welche Wörter sind typischerweise in Spam-Nachrichten?
- Wahrscheinlichkeitsberechnung: Wie oft tauchen bestimmte Wörter in Spam-/Nicht-Spam-E-Mails auf?
- Gewichtung der Merkmale: Welche Eigenschaften weisen am stärksten auf Spam hin?

3️⃣ Anwendung: Klassifikation neuer E-Mails

Neue E-Mail geht ein → Modell analysiert Inhalte → Entscheidung: Spam oder Nicht-Spam?

- 🔹 Falls viele „Spam-indikative“ Wörter enthalten sind → Wahrscheinlichkeit für Spam hoch
- 🔹 Falls seriöse Muster erkannt werden → E-Mail bleibt im Posteingang

4️⃣ Optimierung des Modells

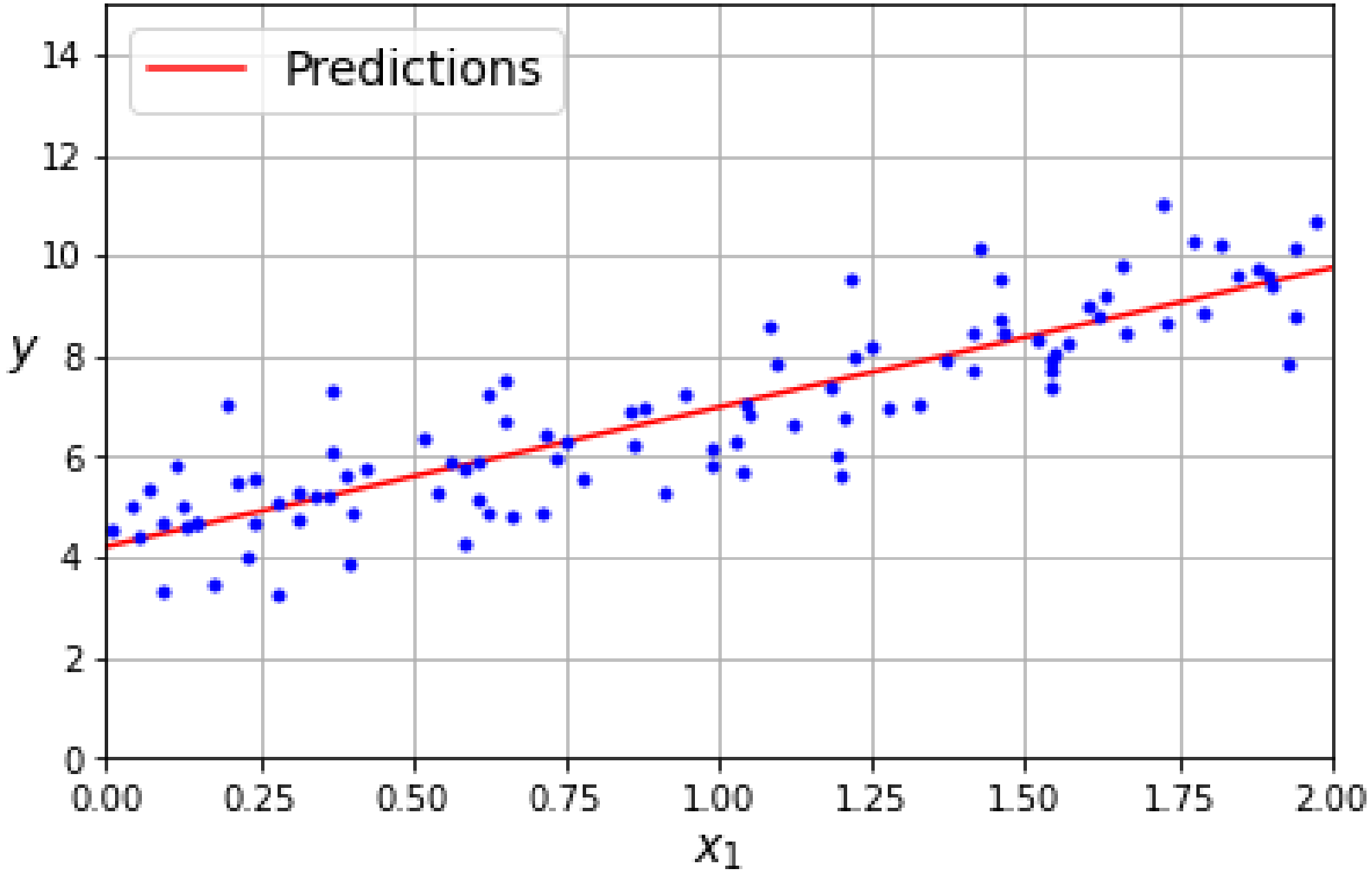
- 🔸 Vermeidung von Fehlern:
  - False Positives 👉 Legitime Mails fälschlicherweise als Spam markiert
  - False Negatives 👉 Spam wird nicht erkannt und landet im Posteingang
    - 🔸 Verbesserung durch kontinuierliches Lernen & Feedback: Nutzer markieren Mails als „Kein Spam“ oder „Als Spam melden“.

📌 Lineare Regression

Die lineare Regression ist ein **grundlegender** (statistischer) Algorithmus, der den Zusammenhang zwischen einer abhängigen Variable (Zielvariable) und einer oder mehreren unabhängigen Variablen (Merkmale/Features) modelliert.

Wichtig für Predictive Analytics:

- Einfachheit und Interpretierbarkeit
- Grundlage für komplexere Modelle
- Schnelles Modell
- Breite Anwendbarkeit


Predictions
14
12
10
8
6
4
2
0
y
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
x_1

Was ist die Idee hinter der linearen Regression?

Sie versucht, eine gerade Linie durch die Datenpunkte zu finden, die den besten Zusammenhang beschreibt.

$$y = m \cdot x + b$$

Bedeutung der Parameter:

✓ x = Eingangsvariable (z. B. Werbebudget)

✓ y = vorhergesagte Zielvariable (z. B. Umsatz)

✓ m = Steigung der Linie (zeigt den Einfluss von x auf y)

✓ b = Achsenabschnitt (Wert von y, wenn $x = 0$)

Mehrfache lineare Regression (multiple regression):

Falls mehrere unabhängige Variablen existieren:

$$y = b + m_1 x_1 + m_2 x_2 + \cdots + m_n x_n$$

- Beispiel: Umsatzvorhersage basierend auf Werbebudget, Anzahl der Geschäfte und Marktanalysen.

Problemformulierung: Modellgleichung

Allgemeine Gleichung der einfachen (univariaten) linearen Regression:

$$y = m \cdot x + b$$

Das Ziel ist, die **besten** Werte für m und b zu finden!

Die gängigste Metrik ist der Mittlere Quadratische Fehler (Mean Squared Error, MSE):

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- y_i = tatsächlicher Wert der i-ten Datenprobe
- $\hat{y}_i$ = vorhergesagter Wert durch das Modell
- n = Anzahl der Datenpunkte

Ziel: Den Fehler so klein wie möglich machen → die Linie passt sich besser an die echten Daten an!

Matrixmultiplikation

- **Definition**:
  - Die Multiplikation zweier Matrizen A (Größe $m \times n$) und B (Größe $n \times p$) ergibt eine neue Matrix C (Größe $m \times p$).
  - Das Element C_{ij} wird berechnet als:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

Beispiel: Matrixmultiplikation

Gegeben:

- Matrix A (2x3):

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- Matrix B (3x2):

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

Schritt-für-Schritt-Berechnung

1. Berechne C_{11}:

$$C_{11} = (1 \cdot 7) + (2 \cdot 9) + (3 \cdot 11) = 7 + 18 + 33 = 58$$

2. Berechne C_{12}:

$$C_{12} = (1 \cdot 8) + (2 \cdot 10) + (3 \cdot 12) = 8 + 20 + 36 = 64$$

3. Berechne C_{21}:

$$C_{21} = (4 \cdot 7) + (5 \cdot 9) + (6 \cdot 11) = 28 + 45 + 66 = 139$$

4. Berechne C_{22}:

$$C_{22} = (4 \cdot 8) + (5 \cdot 10) + (6 \cdot 12) = 32 + 50 + 72 = 154$$

Ergebnis

Die resultierende Matrix C (2x2) ist:

$$C = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

Matrixinversion

- **Definition**:
  - Die Inverse einer quadratischen Matrix A (Größe $n \times n$) ist eine Matrix A^{-1}, sodass:

$$A \cdot A^{-1} = A^{-1} \cdot A = I$$

wobei I die Identitätsmatrix ist:

$$I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ . & . & 1 & . \\ . & . & . & . \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Bedingung:
  - Eine Matrix ist nur invertierbar, wenn ihre Determinante ungleich null ist.

Beispiel: Matrixinversion

Gegeben:

- Matrix A (2x2):

$$A = \begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}$$

Spezialfall für Adjunkte:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, adj(A) = \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Schritt-für-Schritt-Berechnung

1. Berechne die Determinante:

$$\det(A) = (4 \cdot 6) - (7 \cdot 2) = 24 - 14 = 10$$

2. Berechne die Adjunkte:

$$\text{adj}(A) = \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix}$$

3. Berechne die Inverse:

$$A^{-1} = \frac{1}{\det(A)} \cdot \text{adj}(A) = \frac{1}{10} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix} = \begin{bmatrix} 0.6 & -0.7 \\ -0.2 & 0.4 \end{bmatrix}$$

Zusammenfassung

1. **Matrixmultiplikation**:
   - Multipliziere Zeilen der ersten Matrix mit Spalten der zweiten Matrix.

$$A \cdot B = C.$$

2. **Matrixinversion**:
   - Berechne die Inverse einer quadratischen Matrix, falls sie existiert.

$$A^{-1} \cdot A = I.$$

📌 Direkte Lösung der Linearen Regression mittels Normalengleichung
Da die MSE (Mean Squared Error)-Fehlermetrik eine konvexe Funktion ist, kann man ihre Ableitung direkt setzen und nach den optimalen Parametern mm und bb auflösen.

Für die mehrdimensionale lineare Regression lautet die optimale Lösung:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

- $\mathbf{X}$ = Design-Matrix (enthält alle Eingangsvariablen)
- $\mathbf{y}$ = Zielvariablen-Vektor (bekannte Werte)
- $\mathbf{w}$ = gesuchter Parametervektor (enthält Gewichte m und b)

💡 Warum ist das möglich?

- Die Mean-Squared-Error-Kostenfunktion ist eine quadratische Funktion der Parameter, die eine einfache Ableitung erlaubt.
- Weil es eine sog. konvexe Optimierungsaufgabe ist, existiert genau eine eindeutige Lösung, die direkt berechnet werden kann.

📌 Beispiel: Einfache Lineare Regression mit analytischer Lösung

Für eine einfache Regression mit $w_1 = m$ und $w_0 = b$
können wir direkt die sog. Normalengleichung verwenden:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Hiermit erhalten wir die optimalen Werte für m (Steigung) und b (Achsenabschnitt

Vorteile der analytischen Lösung

✅ Exakte Lösung – keine Approximation nötig

✅ Kein Hyperparameter (z. B. Lernrate α) notwendig

✅ Schnell berechenbar für kleine bis mittlere Datensätze

Nachteile der analytischen Lösung

❌ Rechenaufwändig für große Datensätze

Die Matrix-Inversion $(\mathbf{X}^T\mathbf{X})^{-1}$ hat eine Komplexität von $O(n^3)$ → Sehr langsam für Millionen von Datenpunkten.

Alternative: Gradient Descent

Kostenfunktion (Fehlermetrik: Mean Squared Error, MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Hierbei gilt:

y_i = Tatsächlicher Wert

$\hat{y}_i$ = Vorhergesagter Wert anhand der Modellgleichung $\hat{y}_i = mx_i + b$

n = Anzahl der Datenpunkte

Partielle Ableitungen der MSE-Kostenfunktion

Gradientenabstieg benötigt die partiellen Ableitungen des Fehlers nach m (Steigung) und b (Y-Achsenabschnitt)

Partielle Ableitung nach m

$$\frac{\partial}{\partial m} MSE = -\frac{2}{n} \sum_{i=1}^{n} x_i (y_i - \hat{y}_i)$$

Partielle Ableitung nach b

$$\frac{\partial}{\partial b} MSE = -\frac{2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)$$

👉 Diese Ableitungen bestimmen, wie sich m und b in Richtung eines besseren Modells anpassen müssen.

Aktualisierungsregel für den Gradientenabstieg

$$m := m - \alpha \cdot \frac{\partial}{\partial m} MSE$$

$$b := b - \alpha \cdot \frac{\partial}{\partial b} MSE$$

α = Lernrate (Hyperparameter, der bestimmt, wie große Schritte in Richtung der Optimierung gemacht werden)

✔ Subtraktion, weil wir das Minimum des Fehlers suchen

📌 Der Gradientenabstieg nutzt die Ableitung der MSE-Funktion, um m und b schrittweise anzupassen.

📌 Das wiederholt sich so lange, bis die Änderungen minimal sind (Konvergenz). 🚀

Grundprinzip hinter Gradient Descent (GD) und Stochastic Gradient Descent (SGD)

Methode	Berechnet Gradienten auf...	Vorteile	Nachteile
Batch Gradient Descent (BGD)	Allen Datenpunkten	Sehr stabile und genaue Konvergenz	Rechenintensiv bei großen Datenmengen
Stochastic Gradient Descent (SGD)	**Nur einem zufälligen Datenpunkt** pro Schritt	Sehr effizient bei großen Datenmengen	Kann stark schwanken (hohe Varianz)
Mini-Batch Gradient Descent (MBGD)	**Kleine Gruppe von Beispielen (z. B. 32/64 Punkte)**	Balance zwischen GD und SGD, stabil & effizient	Erfordert sorgfältige Wahl der Batch-Größe

📊 Evaluationsmetriken für Regressionsmodelle

📌 1. Mean Absolute Error (MAE)

- Formel:

$$MAE = \frac{1}{m} \sum_{i=1}^{m} |y_i - \hat{y}_i|$$

- Beschreibung:
  - Durchschnittlicher absoluter Fehler zwischen den vorhergesagten Werten $\hat{y}$ und den tatsächlichen Werten y.
- Interpretation:
  - Je **niedriger** der MAE, desto genauer ist das Modell.
  - Beispiel: **MAE = 1000** bedeutet, dass die Vorhersagen im Durchschnitt um **1000** abweichen.

📌 2. Mean Squared Error (MSE)

- Formel:

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

- Beschreibung:
  - Berechnet den **durchschnittlichen quadratischen Fehler**.
  - Höhere Fehler werden stärker gewichtet als bei MAE.
- Interpretation:
  - Niedrigere Werte bedeuten genauere **Modellvorhersagen**.
  - Da Fehler quadriert werden, hat MSE einen größeren Einfluss bei größeren Abweichungen.

📌 3. Root Mean Squared Error (RMSE)

- Formel:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}$$

- Beschreibung:
  - RMSE ist die **Wurzel des MSE**, wodurch der Fehler in der gleichen Einheit wie (y) bleibt.
- Interpretation:
  - **Vergleichbarer mit den tatsächlichen Werten** als MSE.
  - Kleinere Werte deuten auf **besseres Modell** hin.

📌 4. Bestimmtheitsmaß (R^2-Score)

- Formel:

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

  wobei $\bar{y}$ das Mittel der tatsächlichen Werte ist.

- Beschreibung:
  - Misst, wie viel der Variation von y durch die Prädiktoren erklärt wird.
  - Werte liegen zwischen **0 und 1** (oder negativ, wenn das Modell schlechter ist als bloßes Mittelwert-Raten).

- Interpretation:
  - $R^2 = 1$ → Perfektes Modell.
  - $R^2 \approx 0$ → Modell erklärt kaum etwas.
  - $R^2 < 0$ → Modell ist schlechter als eine naive Schätzung.
  - Beispiel: $R^2 = 0.75$ bedeutet, dass **75% der Zielvariablen-Varianz** durch das Modell erklärt wird.

Wichtig

✓ MAE vs. MSE:

- MSE bestraft große Fehler stärker als MAE (weil quadriert).

✓ RMSE:

- Gut interpretierbar, weil gleiche Einheit wie die Zielvariable.

✓ R^2:

- Zeigt die Modellgüte, kann aber täuschen – ein zu hohes R^2 kann auf Overfitting hindeuten!

Immer mehrere Metriken nutzen, um ein Modell korrekt zu bewerten!

📌 Klassifikationsproblem

Die Klassifikation ist eine Problemstellung im überwachten maschinellen Lernen, bei der das Ziel darin besteht, eine Abbildung

$$f : \mathbb{R}^n \to \{C_1, C_2, \ldots, C_k\}$$

zu lernen, die Eingabedaten $\mathbf{x} \in \mathbb{R}^n$ einer von k vordefinierten Klassen C_i zuordnet.

Angenommen, wir haben eine Menge von Trainingsbeispielen:

$$D = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$$

wobei

- $\mathbf{x}_i \in \mathbb{R}^n$ der Feature-Vektor der i-ten Beobachtung ist,
- $y_i \in \{C_1, C_2, ..., C_k\}$ das Label der i-ten Beobachtung ist.

Das Ziel der Klassifikation ist es, eine hypothetische Funktion f zu erlernen:

$$f(\mathbf{x}) = y$$

wobei f die unbekannte wahre Entscheidungsfunktion (Hypothese) ist, die ein ML-Modell approximiert.

Wahrscheinlichkeitsmodell der Klassifikation 📊

Im klassischen ML-Ansatz versucht man, die bedingte Wahrscheinlichkeitsverteilung $P(y|\mathbf{x})$ zu approximieren:

$$P(y = C_k|\mathbf{x}) = f(\mathbf{x})$$

Ein Modell gibt dann die Klasse mit der höchsten Wahrscheinlichkeit aus:

$$\hat{y} = \arg\max_{C_k} P(y = C_k|\mathbf{x})$$

Beispiele für wahrscheinlichkeitsbasierte Modelle:

✓ Logistische Regression
✓ Naive Bayes
✓ Neuronale Netze

Bewertungskriterien für Modelle und Systeme

Genauigkeit (Accuracy)

- Definition:
  - Bezieht sich auf die Bewertung, die angewendet wird, um das qualifizierteste Modell zur Identifizierung von Zusammenhängen in einem Datensatz basierend auf Eingabe- oder Trainingsdaten auszuwählen.
- Formel (Fawcett 2006):

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{True Positives} + \text{False Positives} + \text{True Negatives} + \text{False Negatives}}$$

Aktualität (Timeliness)

- Bezieht sich auf die Verfügbarkeit und Zugänglichkeit von Daten für die Entscheidungen.
- Klare, gut organisierte Daten ermöglichen gute Entscheidungen und ein besseres Verständnis zukünftiger Erwartungen.

Kosten (Cost)

- Der Preis für den Dienst.

Skalierbarkeit (Scalability)

- Bezieht sich auf die Messung, ob ein Algorithmus/Framework/Plattform schnelle Veränderungen im Datenwachstum bewältigen kann.

Zuverlässigkeit (Reliability)

- Bezieht sich auf die Wahrscheinlichkeit, dass ein System eine bestimmte Aufgabe in einer spezifischen Umgebung und zu einer bestimmten Zeit ausführen kann.

Leistung (Performance)

- Die Menge an nützlicher Arbeit, die in einer bestimmten Zeit erledigt wird.

Gültigkeit (Validity)

- Überprüft, ob Modelle wie erwartet und gemäß ihren Designzwecken und geschäftlichen Anwendungen funktionieren.

Ressourcennutzung (Resource Utilization)

- Bezieht sich auf den prozentualen Anteil der Zeit, in der eine Komponente genutzt wird, im Vergleich zur Gesamtzeit, in der die Komponente verfügbar ist.

Zeit

- Faktoren, die sich auf die Zeit beziehen, wie Verarbeitungszeit, Gesamtzeit zur Bereitstellung einer Ausgabe und Ausführungszeit.

Energie

- Die Gesamtmenge an Energie, die verbraucht wird, um die angewandten Anfragen auszuführen.

Throughput

- Die maximale Menge an verarbeiteten Daten in einem System zu einem bestimmten Zeitpunkt.

Nachhaltigkeit (Sustainability)

- Die Fähigkeit des Modells, auf einem bestimmten Niveau gehalten zu werden, ohne zukünftige Updates zu benötigen.

Machbarkeit (Feasibility)

- Die Möglichkeit, dass eine Aussage oder ein Modell komfortabel umgesetzt werden kann.

Sicherheit (Security)

- Der Grad, in dem ein System frei von Bedrohungen ist oder irreparable Konsequenzen vermeidet.
- Besonders kritisch in Smart Cities und im Smart Healthcare.

Präzision (Precision)

- Definition:
  - Die Qualität, der Zustand oder die Tatsache, exakt und präzise zu sein, während ein Modell getestet wird.

2. Datenaufbereitung und -exploration

Datenbeschaffung und -bereinigung

- Datenquellen identifizieren
- Umgang mit fehlenden Werten
- Datenbereinigungstechniken

Datenerkundungstechniken

- Deskriptive Statistiken
- Visualisierungen (Histogramme, Scatterplots)
- Korrelationsanalysen

Feature Engineering: Auswahl und Transformation von Merkmalen

- Auswahl relevanter Features

3. Auswahl und Entwicklung von Vorhersagemodellen mit PyTorch

Einführung in das PyTorch-Framework

- Grundlagen von PyTorch
- Tensoren und ihre Operationen
- Autograd und Differenzierbarkeit

Erstellung von Trainings- und Testdatensätzen

- Datenaufteilung (Trainings-, Validierungs-, Testset)
- Datenladeprozesse mit `DataLoader`

Modellentwicklung und -training mit PyTorch

- Aufbau eines neuronalen Netzwerks
- Trainingsschleife und Verlustfunktion

Was ist PyTorch?

- **PyTorch**: Ein Open-Source-Framework für maschinelles Lernen, entwickelt von Facebook AI Research (FAIR).
- **Einsatzbereiche**:
  - Deep Learning
  - Neuronale Netze
  - Forschung und Produktion
- **Vorteile**:
  - Flexibilität und Benutzerfreundlichkeit
  - Dynamische Berechnungsgraphen
  - Große Community und umfangreiche Bibliotheken

Geschichte von PyTorch

- **Entstehung**:
  - Entwickelt von **Facebook AI Research (FAIR)**.
  - Erstveröffentlichung im **Oktober 2016**.
- **Inspiration**:
  - Basierend auf dem älteren Framework **Torch** (geschrieben in Lua).
  - Ziel: Ein modernes, Python-basiertes Framework für Deep Learning.
- **Wachstum**:
  - Schnelle Adoption in der Forschung und Industrie.
  - Heute eines der **beliebtesten Frameworks** für Deep Learning.

Meilensteine in der Entwicklung von PyTorch

- **2016**:
  - Erste Version von PyTorch veröffentlicht.
  - Fokus auf **Forschung** und **Experimentierfreundlichkeit**.
- **2018**:
  - Einführung von **TorchScript** für die Produktionsbereitstellung.
  - Verbesserte **GPU-Unterstützung** und Performance.
- **2019**:
  - PyTorch 1.0: Kombination von Forschung und Produktion.
  - Integration von **ONNX** (Open Neural Network Exchange) für Modellaustausch.

- **2020**:
  - PyTorch wird zum **Standardframework** in vielen Forschungsbereichen.
  - Einführung von **PyTorch Lightning** für vereinfachtes Modelltraining.
- **2021–2023**:
  - Starke Fokussierung auf **Skalierbarkeit** und **Produktionsreife**.
  - Erweiterte Unterstützung für **Distributed Training** und **Edge Devices**.

Warum wurde PyTorch entwickelt?

- Probleme mit bestehenden Frameworks:
  - Statische Berechnungsgraphen (z. B. TensorFlow) waren unflexibel für Forschung.
  - Komplexität: Viele Frameworks waren schwer zu debuggen und zu erweitern.
- Ziele von PyTorch:
  - Flexibilität: Dynamische Berechnungsgraphen für einfache Experimente.
  - Benutzerfreundlichkeit: Python-first-Design für intuitive Nutzung.
  - Performance: Effiziente Nutzung von GPUs und TPUs.

PyTorch in der Forschung

- **Beliebtheit**:
  - PyTorch ist das **am häufigsten verwendete Framework** in der akademischen Forschung.
  - Viele **State-of-the-Art-Modelle** (z. B. Transformer, GPT, BERT) wurden mit PyTorch entwickelt.
- **Vorteile für Forscher**:
  - **Dynamische Graphen**: Einfache Modifikationen während des Trainings.
  - **Debugging**: Direkte Integration mit Python-Debugging-Tools.
  - **Community**: Große, aktive Community und umfangreiche Dokumentation.

PyTorch in der Industrie

- Einsatzbereiche:
  - Predictive Analytics
  - Computer Vision: Bilderkennung, Objekterkennung.
  - Natural Language Processing (NLP): Sprachmodelle, Übersetzung.
- Unternehmen, die PyTorch nutzen:
  - Facebook (Meta): Für KI-Forschung und Produkte.
  - Tesla: Für autonomes Fahren.
  - OpenAI: Für die Entwicklung von GPT-Modellen.
  - Uber, Airbnb, Microsoft: Für verschiedene KI-Anwendungen.

PyTorch vs. Andere Frameworks

- TensorFlow:
  - Vorteile: Bessere Produktionsreife, TensorFlow Serving.
  - Nachteile: Komplexität, statische Graphen.
- Keras:
  - Vorteile: Einfachheit, hohe Abstraktion.
  - Nachteile: Weniger flexibel für komplexe Modelle.
- scikit-learn:
  - Vorteile: Umfassende Bibliothek für klassische Algorithmen
  - Nachteile: Keine Deep Learning-Unterstützung

Wann PyTorch verwenden?

- **Forschung**:
  - Dynamische Graphen ermöglichen einfache Experimente.
  - Ideal für **State-of-the-Art-Modelle** und **akademische Projekte**.
- **Prototyping**:
  - Schnelle Iteration und einfaches Debugging.
- **Produktion**:
  - Mit **TorchScript** und **ONNX** ist PyTorch auch für Produktionsumgebungen geeignet.

Wann TensorFlow verwenden?

- Produktion:
  - TensorFlow Serving und TFX bieten robuste Lösungen für die Bereitstellung.
- Skalierbarkeit:
  - Unterstützung für verteiltes Training auf großen Clustern.
- Edge Devices:
  - TF Lite für mobile und eingebettete Systeme.

Wann Keras verwenden?

- **Einfache Modelle**:
  - Ideal für schnelle Prototypen und einfache Anwendungen.
- **Einsteiger**:
  - Niedrige Einstiegshürde durch hohe Abstraktion.
- **Integration mit TensorFlow**:
  - Keras läuft nahtlos auf TensorFlow und nutzt dessen Infrastruktur.

Wann scikit-learn verwenden?

- **Klassisches Machine Learning**:
  - Algorithmen wie lineare Regression, Entscheidungsbäume, SVM, Clustering usw.
- **Kleine bis mittelgroße Datensätze**:
  - Ideal für Tabellendaten und traditionelle ML-Probleme.
- **Schnelle Experimente**:
  - Einfache API für schnelles Prototyping und Evaluierung.

Kann gut für das Predictive Analytics geeignet sein!

Tensoren: Grundlegende Datenstruktur in PyTorch

Was ist ein Tensor?

- **Definition**:
  - Ein **Tensor** ist ein mehrdimensionales Array, ähnlich wie NumPy-Arrays.
  - Kann skalare Werte, Vektoren, Matrizen oder höherdimensionale Daten speichern.
- **Beispiele**:
  - **Skalar**: Ein einzelner Wert (z. B. `3.14`).
  - **Vektor**: Eine eindimensionale Liste von Werten (z. B. `[1, 2, 3]`).
  - **Matrix**: Eine zweidimensionale Tabelle von Werten (z. B. `[[1, 2], [3, 4]]`).
  - **Höherdimensionale Tensoren**: 3D, 4D usw. (z. B. `[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]`).

Eigenschaften von Tensoren

- Datenstruktur:
  - Speichert **numerische Daten** (z. B. `float`, `int`, `double`).
  - Unterstützt **verschiedene Datentypen** (dtypes).
- Operationen:
  - **Mathematische Operationen**: Addition, Subtraktion, Multiplikation, Division.
  - **Matrixoperationen**: Matrixmultiplikation, Transponierung, Inversion.
  - **Reduktionsoperationen**: Summe, Mittelwert, Maximum, Minimum.
- GPU-Unterstützung:
  - Tensoren können auf **GPUs** verschoben werden, um Berechnungen zu beschleunigen.
  - Ermöglicht **paralleles Rechnen** für Deep Learning.

Tensor-Erstellung in PyTorch

- Beispiele:

```
import torch

# Erstellen eines Tensors aus einer Python-Liste
tensor = torch.tensor([1, 2, 3])

# Erstellen eines zufälligen Tensors
random_tensor = torch.rand(2, 3)  # 2x3 Matrix mit Zufallswerten

# Erstellen eines Tensors mit Nullen
zeros_tensor = torch.zeros(3, 3)  # 3x3 Matrix mit Nullen

# Erstellen eines Tensors mit Einsen
ones_tensor = torch.ones(2, 2)  # 2x2 Matrix mit Einsen
```

```python
# Addition
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
c = a + b  # Ergebnis: tensor([5, 7, 9])

# Matrixmultiplikation
mat1 = torch.tensor([[1, 2], [3, 4]])
mat2 = torch.tensor([[5, 6], [7, 8]])
result = torch.matmul(mat1, mat2)  # Ergebnis: tensor([[19, 22], [43, 50]])

# Summe aller Elemente
tensor = torch.tensor([[1, 2], [3, 4]])
sum_all = tensor.sum()  # Ergebnis: 10

# Mittelwert aller Elemente
mean_all = tensor.mean()  # Ergebnis: 2.5
```

GPU-Unterstützung für Tensoren

```
# Prüfen, ob eine GPU verfügbar ist
if torch.cuda.is_available():
    device = torch.device('cuda')  # GPU
else:
    device = torch.device('cpu')   # CPU

# Tensor auf die GPU verschieben
tensor_gpu = tensor.to(device)
```

Tensoren vs. NumPy-Arrays

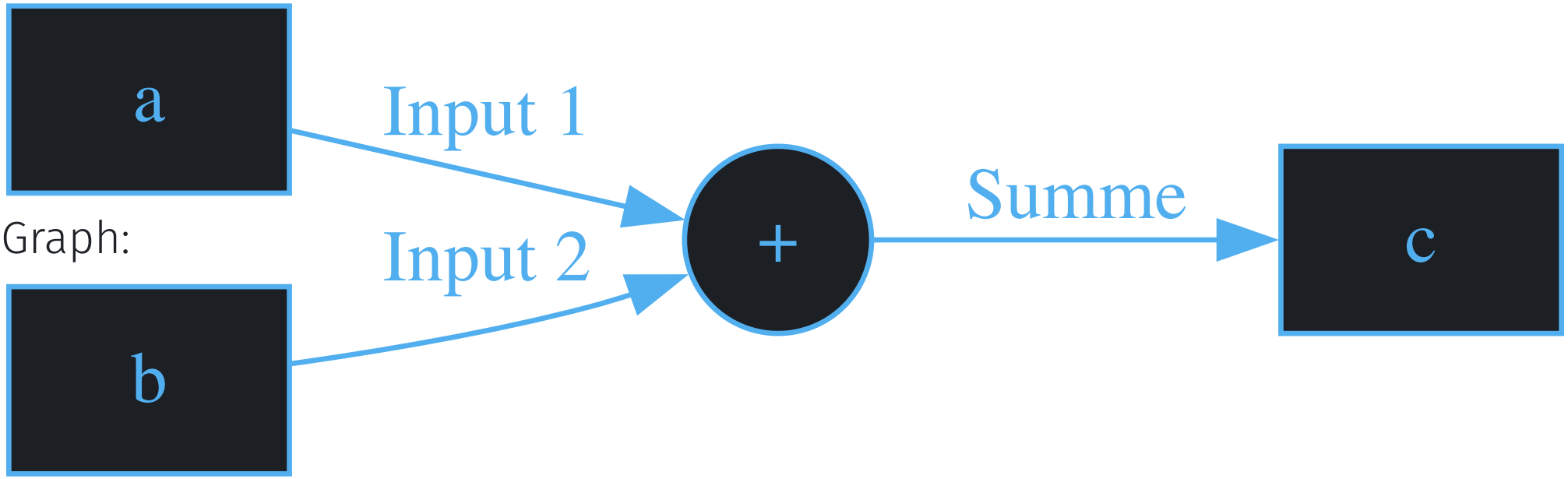
- Ähnlichkeiten:
  - Beide sind mehrdimensionale Arrays.
  - Ähnliche API für viele Operationen.
- Unterschiede:
  - GPU-Unterstützung: PyTorch-Tensoren können auf GPUs verschoben werden.
  - Autograd: PyTorch-Tensoren unterstützen automatische Differenzierung (für Deep Learning).
  - Dynamische Graphen: PyTorch-Tensoren sind Teil von dynamischen Berechnungsgraphen.

Dynamische Berechnungsgraphen in PyTorch

Was ist ein Berechnungsgraph?

- Definition:
  - Ein **Berechnungsgraph** ist eine Darstellung von mathematischen Operationen als Knoten und Datenflüsse als Kanten.
  - Wird verwendet, um komplexe Berechnungen zu modellieren und zu optimieren.
- Beispiel:
  - Operation: `c = a + b`

- Graph:



Statische vs. Dynamische Graphen

- **Statische Graphen** (z. B. TensorFlow 1.x):
  - Der Graph wird **einmalig definiert** und dann ausgeführt.
  - **Vorteile**:
    - Effiziente Optimierung und Ausführung.
  - **Nachteile**:
    - Unflexibel für Experimente und Debugging.
    - Schwer zu modifizieren während der Laufzeit.

Dynamische Graphen (PyTorch):

- Der Graph wird **zur Laufzeit** (on fly) aufgebaut.
- **Vorteile**:
  - Flexibilität: Einfache Modifikationen während des Trainings.
  - Ideal für **Prototyping** und **Iteration** (Train->Test->Adapt->(Deploy)).
  - Benutzer können Modelle **dynamisch anpassen** (z. B. RNNs mit variabler Länge).
  - Da der Graph zur Laufzeit erstellt wird, können Python-Debugging-Tools wie `pdb` verwendet werden.
- **Nachteile**:
  - Geringfügig weniger effizient als statische Graphen.

```python
import tensorflow as tf

# Graph definieren
x = tf.placeholder(tf.float32, shape=(None, 10))  # Eingabe
W = tf.Variable(tf.random_normal([10, 1]))        # Gewichte
b = tf.Variable(tf.zeros([1]))                    # Bias
y = tf.matmul(x, W) + b                           # Ausgabe

# Session starten und Graph ausführen
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())  # Variablen initialisieren
    result = sess.run(y, feed_dict={x: [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]})
    print(result)
```

```python
import torch
import torch.nn as nn

class DynamicNet(nn.Module):
    def __init__(self, num_layers):
        super(DynamicNet, self).__init__()
        self.layers = nn.ModuleList([nn.Linear(10, 10) for _ in range(num_layers)])

    def add_layer(self, input_size, output_size):
        """Fügt eine neue Schicht hinzu."""
        self.layers.append(nn.Linear(input_size, output_size))

    def forward(self, x):
        for layer in self.layers:
            x = torch.relu(layer(x))
        return x

# Modell mit 3 Schichten erstellen
model = DynamicNet(num_layers=3)
input_data = torch.randn(1, 10)
output = model(input_data)
print(output)
```

```
# Modell erstellen
model = DynamicNet()

# Erste Schicht hinzufügen
model.add_layer(input_size=10, output_size=10)

# Eingabedaten
input_data = torch.randn(1, 10)

# Vorwärtsdurchlauf mit einer Schicht
output = model(input_data)
print("Ausgabe mit 1 Schicht:\n", output)

# Zweite Schicht hinzufügen
model.add_layer(input_size=10, output_size=10)

# Vorwärtsdurchlauf mit zwei Schichten
output = model(input_data)
print("Ausgabe mit 2 Schichten:\n", output)

# Dritte Schicht hinzufügen
model.add_layer(input_size=10, output_size=10)

# Vorwärtsdurchlauf mit drei Schichten
output = model(input_data)
print("Ausgabe mit 3 Schichten:\n", output)
```

Autograd

- **Definition**:
  - **Autograd** ist das automatische Differenzierungssystem in PyTorch.
  - Es berechnet **Gradienten** von Tensoren automatisch, was für das Training von neuronalen Netzen entscheidend ist.
- **Funktionsweise**:
  - PyTorch zeichnet alle Operationen auf einem Tensor auf (Forward Pass).
  - Beim Backward Pass werden die Gradienten mithilfe der **Kettenregel** berechnet.

Differenzierbarkeit in PyTorch

- Differenzierbare Tensoren:
  - Ein Tensor wird differenzierbar, wenn `requires_grad=True` gesetzt wird.
  - PyTorch verfolgt dann alle Operationen auf diesem Tensor, um den Gradienten zu berechnen.

Forward Pass:

PyTorch zeichnet alle Operationen in einem Computational Graph auf.
Der Graph speichert die Operationen: Quadrieren, Multiplizieren, Addieren.

Backward Pass:

PyTorch verwendet die Kettenregel, um den Gradienten zu berechnen.

- Beispiel:

```python
import torch

# Erstelle einen Tensor mit requires_grad=True
x = torch.tensor(2.0, requires_grad=True)

# Definiere eine Funktion
y = x**2 + 3*x + 1

# Berechne den Gradienten
y.backward()

# Gradient von y bezüglich x
print(x.grad)  # Ausgabe: 7.0
```

Anwendung in NN

```
# Definiere ein einfaches Modell
model = nn.Linear(10, 1)

# Definiere die Verlustfunktion und den Optimierer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Forward Pass
output = model(torch.randn(10))
target = torch.randn(1)
loss = criterion(output, target)

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

Aufteilung der Daten

- Ziel:
  - Vermeidung von **Overfitting** (Überanpassung) und **Underfitting** (Unteranpassung).
  - Bewertung der **Generalisierung** des Modells auf neue, unbekannte Daten.
- **Aufteilung**:
  - **Trainingsset**: Zum Trainieren des Modells.
  - **Validierungsset**: Zum Anpassen der Hyperparameter und zur Bewertung während des Trainings.
  - **Testset**: Zur endgültigen Bewertung des Modells nach dem Training.

Typische Aufteilung

- Trainingsset: 60-80% der Daten.
- Validierungsset: 10-20% der Daten.
- Testset: 10-20% der Daten.

Datenaufteilung in PyTorch

- Bibliotheken:
  - `torch.utils.data.random_split`: Zum Aufteilen eines Datensatzes.
  - `torch.utils.data.DataLoader`: Zum Laden der Daten in Batches.

Weitere wichtige Kostenfunktionen für DNNs

Binary Cross-Entropy

- Die Binary Cross-Entropy (BCE) ist eine Verlustfunktion für **binäre Klassifikationsprobleme**.
- Sie misst die Differenz zwischen den **vorhergesagten Wahrscheinlichkeiten** und den **tatsächlichen Labels**.

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)\right]$$

- y_i: Tatsächliches Label (0 oder 1).
- p_i: Vorhergesagte Wahrscheinlichkeit (zwischen 0 und 1).
- N: Anzahl der Samples.

Warum BCE?

- Eigenschaften:
  - Straft große Abweichungen zwischen Vorhersage und tatsächlichem Label stark.
  - Gut geeignet für Probleme mit zwei Klassen (z. B. Churn-Vorhersage, Spam-Erkennung).
- Vorteile:
  - Einfach zu berechnen.
  - Gut interpretierbar.
  - Funktioniert gut mit Sigmoid-Aktivierungsfunktionen.

Beispiel

- Tatsächliche Labels:

$$y = [1, 0, 1, 1]$$

- Vorhergesagte Wahrscheinlichkeiten:

$$p = [0.9, 0.2, 0.8, 0.4]$$

- BCE berechnen:

$$\text{BCE} = -\frac{1}{4}\left[1 \cdot \log(0.9) + 1 \cdot \log(0.8) + 1 \cdot \log(0.8) + 1 \cdot \log(0.4)\right] \approx 0.3667.$$

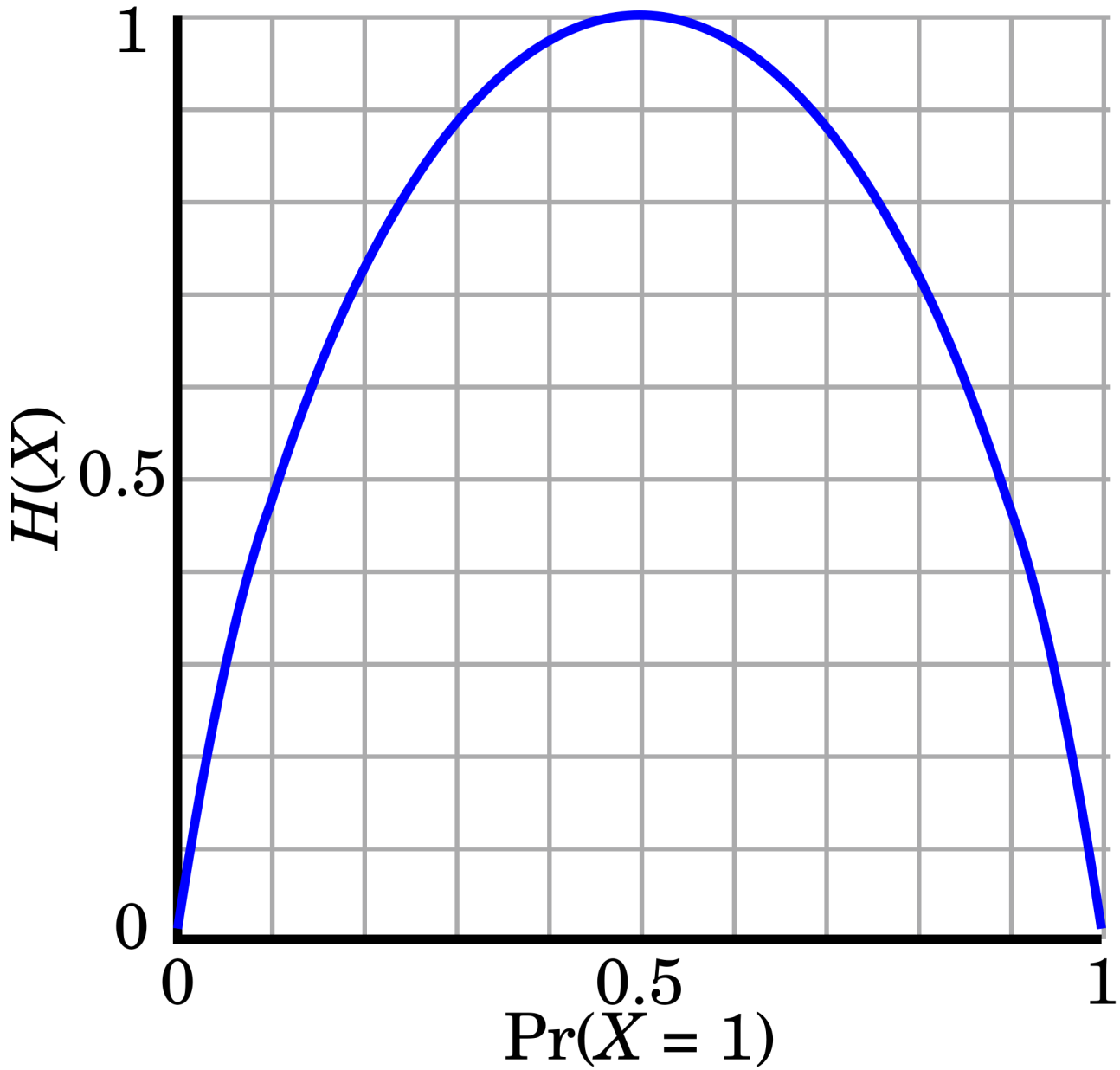
relativ niedrig, d.h. die Vorhersagen sind akzeptabel

Anwendung in PyTorch

```
import torch
import torch.nn as nn

# Beispiel: Vorhergesagte Wahrscheinlichkeiten und tatsächliche Labels
predictions = torch.tensor([0.9, 0.2, 0.8, 0.4])
labels = torch.tensor([1.0, 0.0, 1.0, 1.0])

# Binary Cross-Entropy Loss
criterion = nn.BCELoss()
loss = criterion(predictions, labels)
print(f"BCE Loss: {loss.item()}")
```


1
0.5
0
H(X)
0
0.5
1
Pr(X = 1)

Was ist Cross-Entropy?

- Die Cross-Entropy misst die Differenz zwischen zwei Wahrscheinlichkeitsverteilungen.
- In Machine Learning wird sie verwendet, um die Differenz zwischen den **vorhergesagten Wahrscheinlichkeiten** und den **tatsächlichen Labels** zu messen.

$$\text{Cross-Entropy} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} \cdot \log(p_{ij})$$

- y_{ij}: Tatsächliches Label (One-Hot-Encoded).
- p_{ij}: Vorhergesagte Wahrscheinlichkeit für Klasse (j).
- N: Anzahl der Samples, C: Anzahl der Klassen.

- Eigenschaften:
  - Straft große Abweichungen zwischen Vorhersage und tatsächlichem Label stark.
  - Gut geeignet für **Multi-Klassen-Klassifikation**.
- Vorteile:
  - Einfach zu berechnen.
  - Gut interpretierbar.
  - Funktioniert gut mit Softmax-Aktivierungsfunktionen.

Anwendung in PyTorch

```
import torch
import torch.nn as nn

# Beispiel: Vorhergesagte Wahrscheinlichkeiten und tatsächliche Labels
predictions = torch.tensor([[0.9, 0.1, 0.0], [0.2, 0.7, 0.1], [0.1, 0.2, 0.7], [0.8, 0.1, 0.1]])
labels = torch.tensor([[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 0, 0]])

# Cross-Entropy Loss
criterion = nn.CrossEntropyLoss()
loss = criterion(predictions, labels)
print(f"Cross-Entropy Loss: {loss.item()}")
```

4. Praktische Übungen zur Implementierung von Deep Learning-Modellen

Hands-on-Übungen zur Implementierung von Deep Learning-Modellen in PyTorch

- Aufbau eines einfachen neuronalen Netzwerks
- Training auf Beispiel-Datensätzen

Trainieren von Modellen anhand von realen Daten

- Anwendung auf echte Datensätze
- Umgang mit komplexeren Datenstrukturen

Recurrent Neural Network

- Ein RNN ist ein neuronales Netzwerk, das für die Verarbeitung von **sequenziellen Daten** entwickelt wurde.
    - Es hat eine **Gedächtnisfunktion**, die es ermöglicht, Informationen aus früheren Schritten zu speichern.
- **Anwendung**:
    - Zeitreihenvorhersage (z. B. Aktienkurse, Wetter).
    - Textverarbeitung (z. B. Textgenerierung, Sentiment-Analyse).
    - Spracherkennung.

Aufbau eines RNN

- Ein RNN besteht aus einer **wiederholten Zelle**, die Informationen über die Zeit weiterreicht.
    - Jede Zelle nimmt zwei Eingaben:
        a. Den aktuellen Eingabewert x_t.
        b. Den versteckten Zustand h_{t-1} aus dem vorherigen Schritt.

$$h_t = \tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

$$y_t = W_y \cdot h_t + b_y$$

Problem herkömmlicher RNNs

- Vanishing Gradient:
  - Bei langen Sequenzen "verschwinden" die Gradienten während des Backpropagation.
  - Dies führt dazu, dass das Netzwerk nicht lernt.
- Exploding Gradient:
  - Die Gradienten können auch explodieren, was zu instabilen Updates führt.

LSTM

- Long short-term memory
- LSTM ist eine spezielle Art von **Recurrent Neural Network (RNN)**.
- Es wurde entwickelt, um das Problem des **vanishing gradient** in herkömmlichen RNNs zu lösen.
- **Anwendung**:
  - Zeitreihenvorhersage (z. B. Aktienkurse, Wetter).
  - Sequenz-zu-Sequenz-Modelle (z. B. Maschinelle Übersetzung, Textgenerierung).

LSTM vs. RNNs

- **Vanishing Gradient**:
  - Bei langen Sequenzen "verschwinden" die Gradienten während des Backpropagation.
- **Lösung**:
  - LSTM führt **Gedächtniszellen (Memory Cells)** ein, die Informationen über lange Zeiträume speichern können.

Aufbau einer LSTM-Zelle

1. **Forget Gate**: Entscheidet, welche Informationen verworfen werden.
2. **Input Gate**: Fügt neue Informationen hinzu.
3. **Output Gate**: Steuert, welche Informationen ausgegeben werden.

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
\tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
C_t &= f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t &= o_t \cdot \tanh(C_t)
\end{aligned}
$$

Übersicht der Komponenten

Komponente	Formel	Funktion
Forget Gate	$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$	Entscheidet, welche Informationen aus C_{t-1} verworfen werden.
Input Gate	$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$	Entscheidet, welche neuen Informationen zu C_t hinzugefügt werden.
Kandidatenzustand	$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$	Berechnet potenzielle neue Informationen für C_t.

Komponente	Formel	Funktion
Zellzustand	$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$	Aktualisiert den Zellzustand.
Output Gate	$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$	Entscheidet, welche Informationen aus C_t als h_t ausgegeben werden.
Versteckter Zustand	$h_t = o_t \cdot \tanh(C_t)$	Gibt die gefilterte Version des Zellzustands aus.

Erklärung der Symbole

- h_{t-1}: Versteckter Zustand aus dem vorherigen Zeitschritt.
- x_t: Eingabe zum aktuellen Zeitschritt.
- W_f, W_i, W_C, W_o: Gewichtsmatrizen für die jeweiligen Gates.
- b_f, b_i, b_C, b_o: Bias-Werte für die jeweiligen Gates.
- σ: Sigmoid-Funktion (Werte zwischen 0 und 1).
- $\tanh$: Tangens hyperbolicus (Werte zwischen -1 und 1).
- C_t: Aktualisierter Zellzustand.
- h_t: Versteckter Zustand zum aktuellen Zeitschritt.

Datenfluss

1. **Eingabe**: h_{t-1} und x_t.
2. **Forget Gate**: Bestimmt, was aus C_{t-1} behalten wird.
3. **Input Gate**: Bestimmt, welche neuen Informationen zu C_t hinzugefügt werden.
4. **Zellzustand**: C_t wird aktualisiert.
5. **Output Gate**: Bestimmt, was aus C_t als h_t ausgegeben wird.
6. **Ausgabe**: h_t wird an den nächsten Zeitschritt weitergegeben.