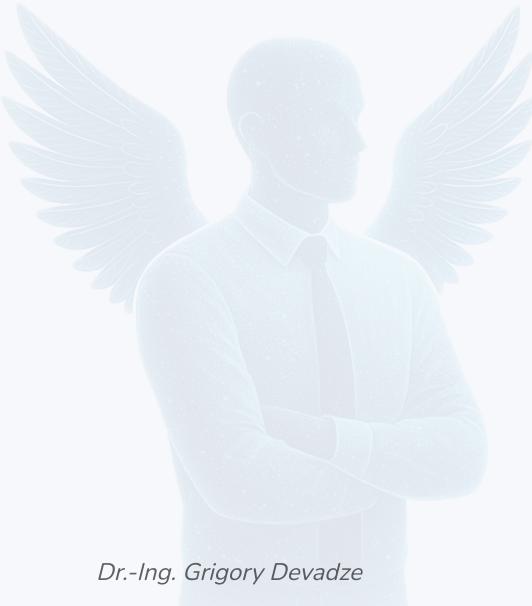


Deep Learning Recap

Dr.-Ing. Grigory Devadze

2025-12-09



Agenda

- Tag 1 Recap & Generative Adversarial Networks (GANs)
- Tag 2 Recurrent Neural Networks (RNNs)
 - RNN
 - LSTM (long short-term memory)
 - GRU (gated recurrent unit)
- Tag 3 Transformer
 - Encoder-decoder
 - Encoder-only
 - Decoder-only

- Tag 4 Large Language Models (LLMs)
- Tag 5 Übungen & Ausblick

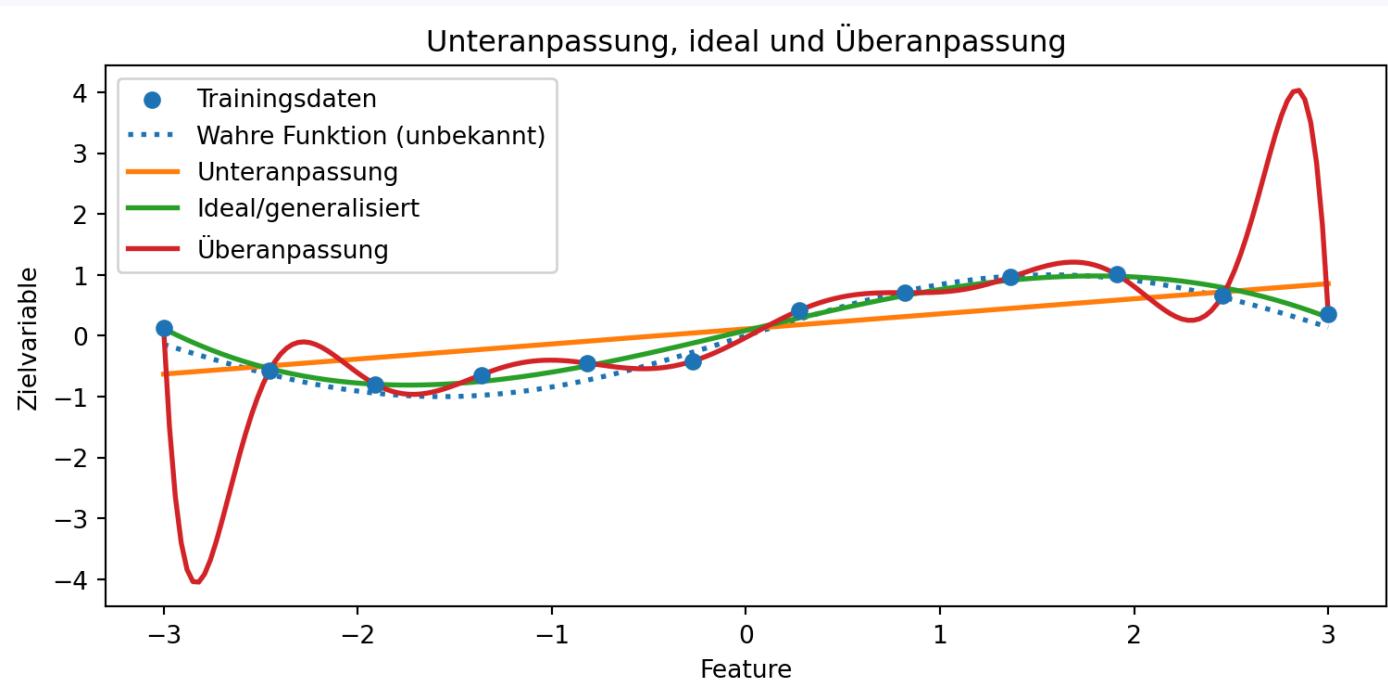


Maschinelles Lernen

- Maschinelles Lernen findet Regeln/Funktionen, die Eingabedaten auf gewünschte Ausgaben abbilden – ohne dass wir den exakten Algorithmus vorgeben.
- Wir trainieren ein Modell auf Beispelpaaren: (Eingabe, erwartete Ausgabe).
- **Überwachtes Lernen (Supervised Learning)** = Lernen aus gelabelten Daten.
- Die unbekannte „wahre“ Funktion wird durch ein Modell angenähert, das wir auf Daten fitten.

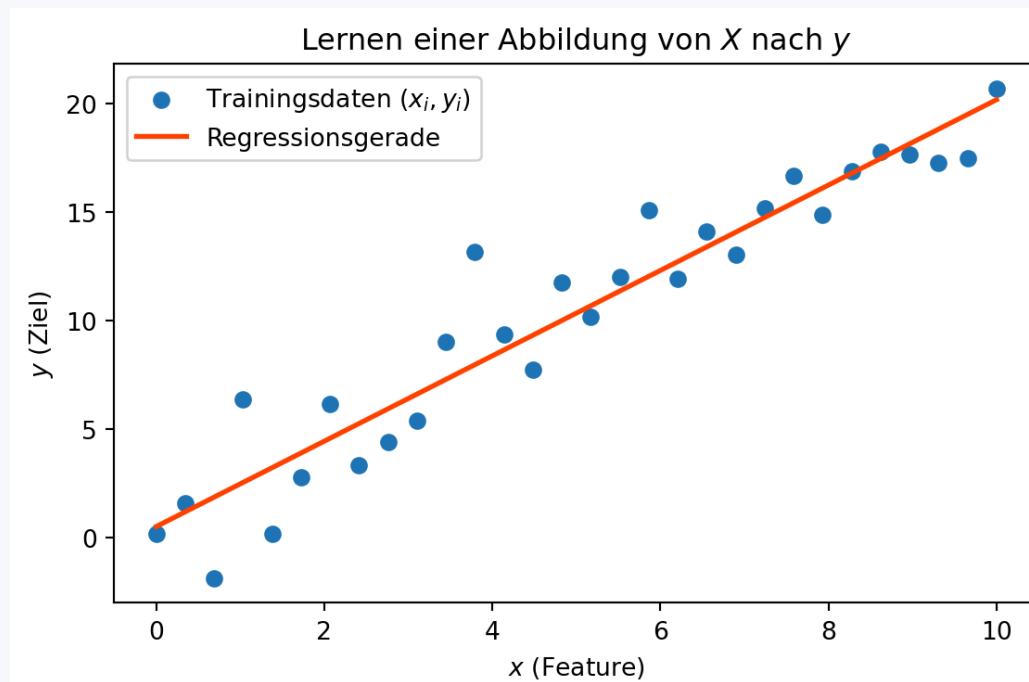
Overfitting und Underfitting

- **Underfitting:** Modell ist zu einfach – verpasst Muster.
- **Overfitting:** Modell ist zu komplex – merkt sich Trainingsdaten, versagt bei neuen Daten.
- **Generalisation:** Ziel ist ein niedriger Fehler auf neuen/ungesehenen Daten.
- Wir steuern die Modellkomplexität („Kapazität“) und nutzen Feature-Engineering sowie mehr Daten.



Maschinelles Lernen

- **Maschinelles Lernen** lernt eine Funktion, die Eingabefeatures X auf Ziel y abbildet.
- **Lineare Regression** modelliert $y = w^\top x + b + \epsilon$, wobei w Gewichte, b Bias und ϵ Rauschen sind.



Verlustminimierung (Loss Optimization)

- Ziel: Finde w, b , sodass das Modell die Trainingsdaten gut erklärt **und** gut generalisiert.
- **MLE/Verlust-Minimierung:** Häufig minimieren wir die mittlere quadratische Abweichung (MSE):


$$L(w, b) = \frac{1}{N} \sum_{i=1}^N (y_i - (w^\top x_i + b))^2$$

Training per Gradientabstieg

- **Gradientabstieg** aktualisiert w, b iterativ in Richtung fallender Loss.
- **Mini-Batch-Gradientabstieg** nutzt kleine Batches als Schätzer des Gesamtgradienten – effizient und robust.



Training

- In jedem Schritt werden die Parameter wie folgt aktualisiert:

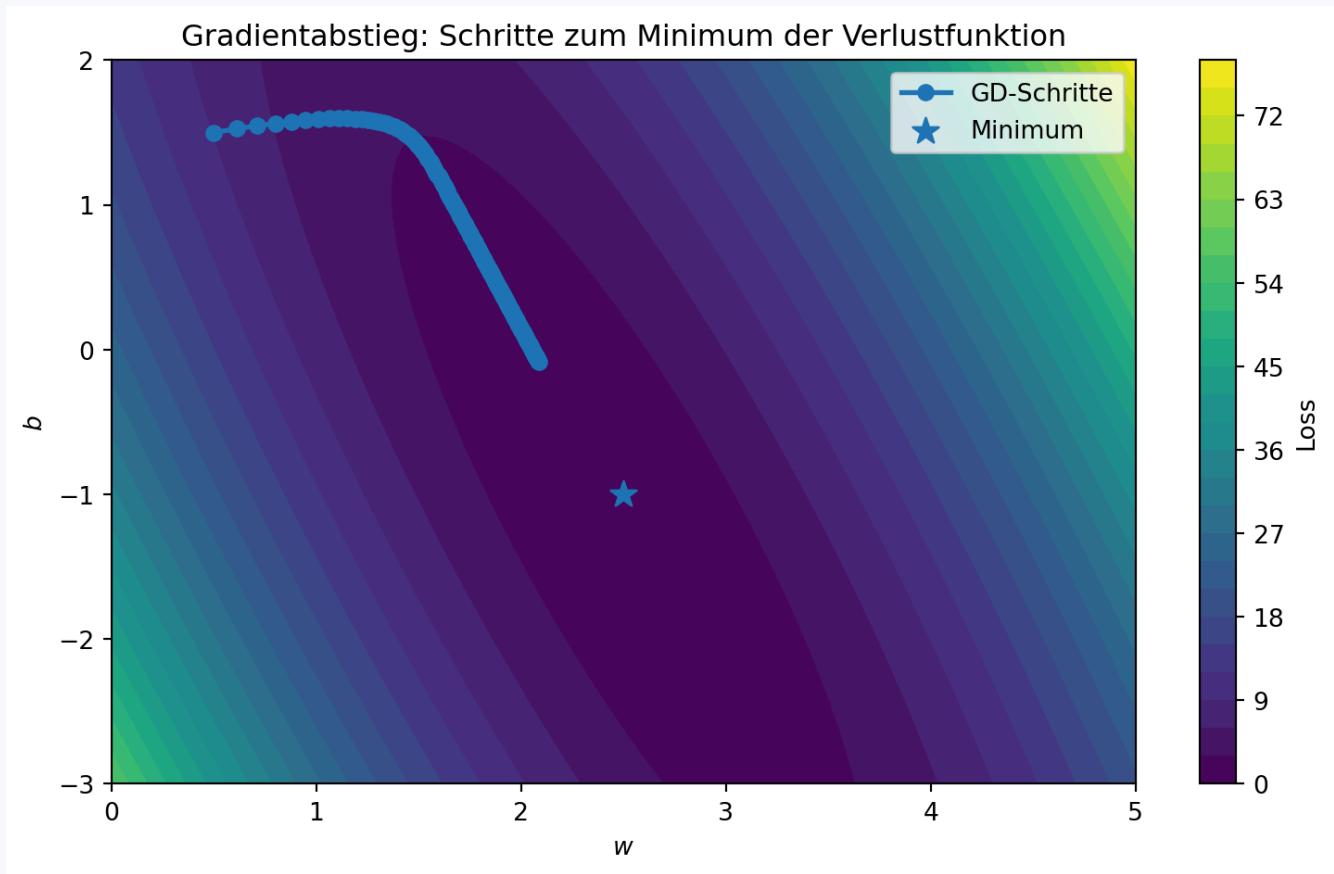
$$w \leftarrow w - \eta \frac{\partial L_{\text{batch}}}{\partial w}$$

$$b \leftarrow b - \eta \frac{\partial L_{\text{batch}}}{\partial b}$$

- η ist die Lernrate; L_{batch} ist der Verlust über einen zufällig gezogenen Mini-Batch.

- Diese Updates werden wiederholt, bis ein (lokales) Minimum der Loss erreicht ist.





Von Linearer Regression zu Neuronalen Netzen

- **Lineare Regression:** lernt eine einfache Funktion

$$y = w^\top x + b$$

- Erfasst nur lineare Zusammenhänge
- Begrenzte Flexibilität; komplexe, nichtlineare Muster bleiben unmodelliert

- **Neuronale Netze:** erweitern dies um Schichten („Neuronen“) und nichtlineare Aktivierungen
 - Können hochkomplexe Funktionen approximieren
 - Erlauben Lernen aus reichhaltigen, strukturierten Daten



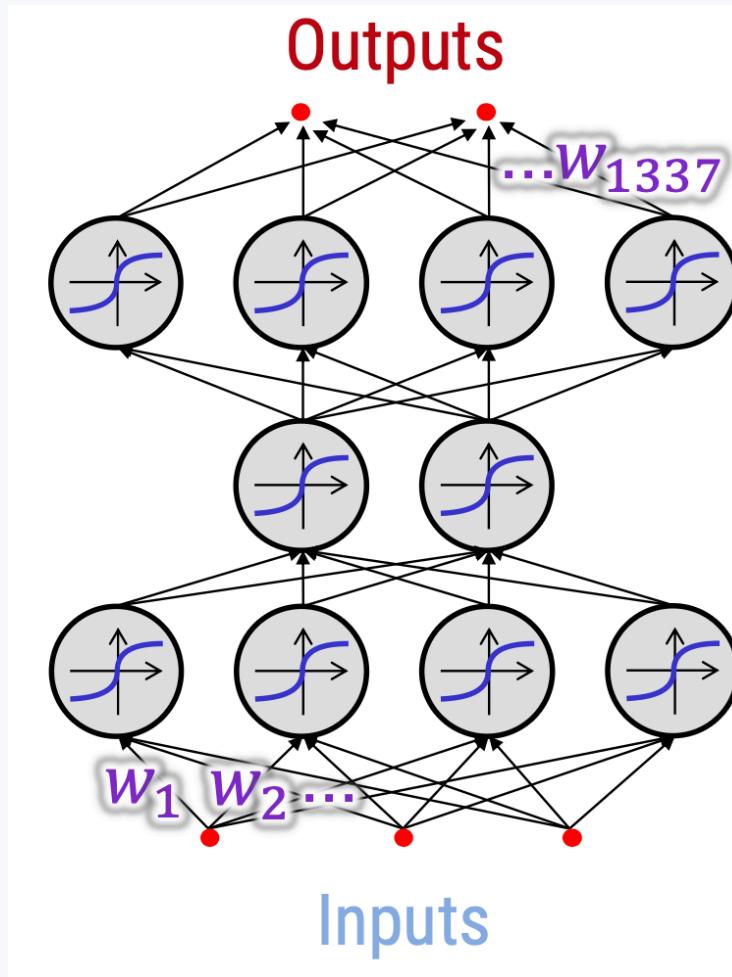
Neuronales Netz als verallgemeinerte Regression

- Ein neuronales Netz lässt sich als **Stack von Regressionen** mit nichtlinearen Transformationen je Schicht betrachten.

$$y = f^{(N)} \left(W^{(N)} f^{(N-1)} \left(\cdots f^{(1)}(W^{(1)}x + b^{(1)}) \cdots \right) + b^{(N)} \right)$$

- Dabei gilt:
 - x = Eingabefeatures
 - $W^{(\ell)}, b^{(\ell)}$ = Gewichte und Bias je Schicht ℓ
 - $f^{(\ell)}$ = Aktivierungsfunktion (z. B. ReLU, Sigmoid)
 - N = Anzahl der Schichten (Tiefe)



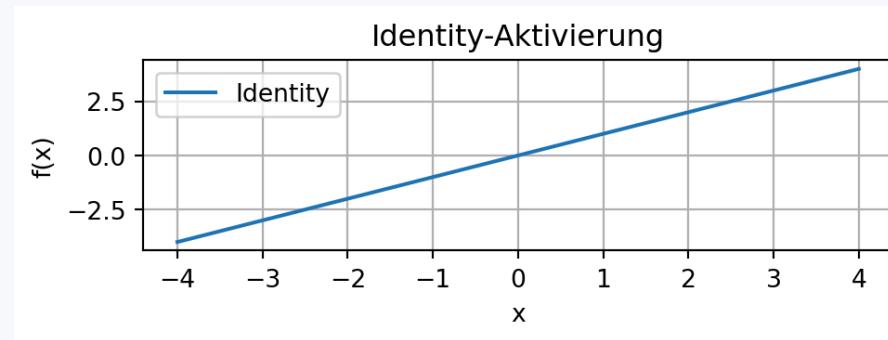


Aktivierungsfunktionen in Neuronalen Netzen

- Aktivierungsfunktionen bringen Nichtlinearität und Flexibilität.
- Häufige Varianten: **Identity, ReLU, Sigmoid, tanh, Softmax**
- Die passende Wahl hängt von Aufgabe und Schicht ab.

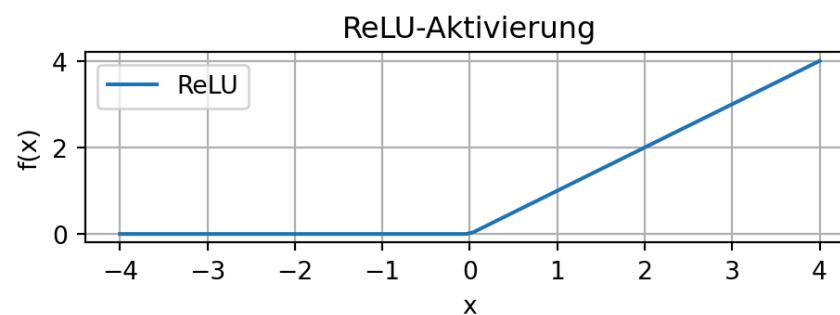
Identity (Identität)

- Formel: $f(x) = x$
- Linear, keine Nichtlinearität
- Z. B. Ausgabeschicht bei Regression (unbeschränkt)



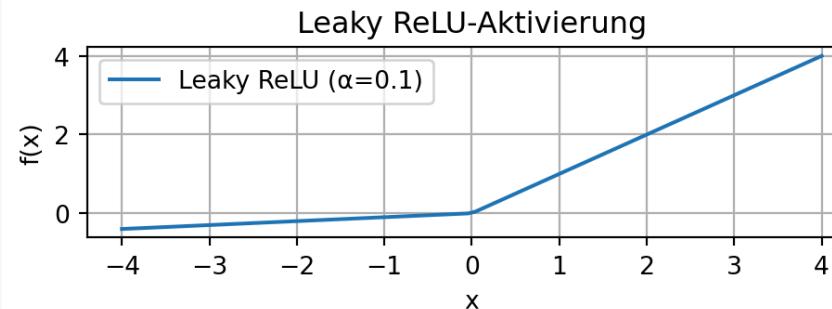
Rectified Linear Unit (ReLU)

- Formel: $f(x) = \max(0, x)$
- Beliebt in Hidden-Schichten tiefer Netze; effizient, reduziert Vanishing Gradients



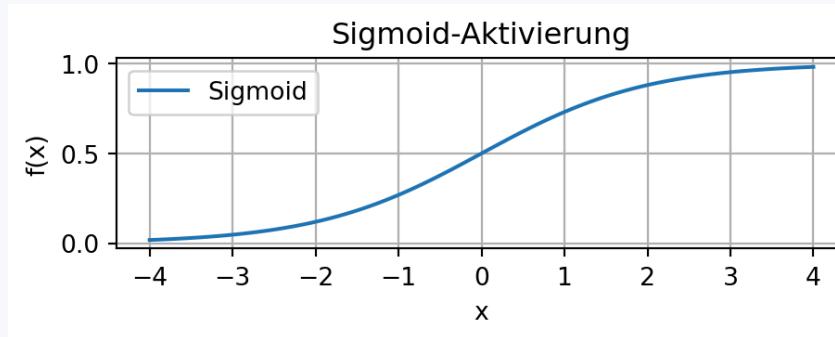
Leaky Rectified Linear Unit (Leaky ReLU)

- Formel: $f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$ mit z.B. $\alpha = 0.01$
- Löst das „tote Neuronen“-Problem klassischer ReLU
- Negative Werte werden mit kleinem Faktor weitergegeben



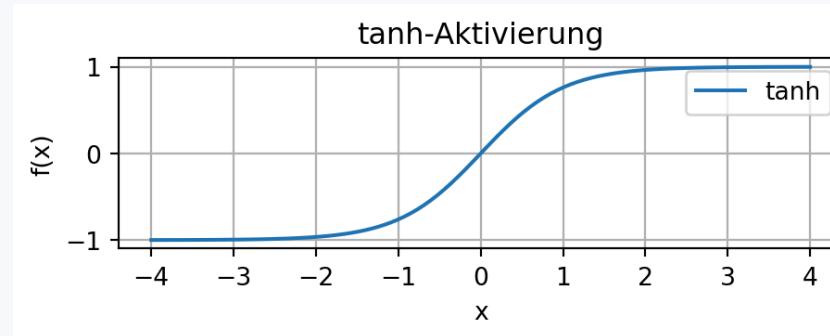
Sigmoid

- Formel: $f(x) = \frac{1}{1+e^{-x}}$
- Ausgabe zwischen 0 und 1 (Wahrscheinlichkeit)
- In der binären Ausgabeschicht gebräuchlich



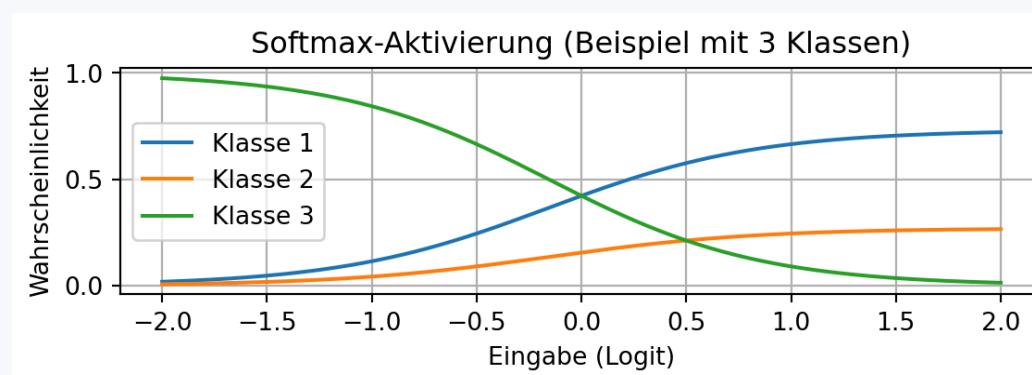
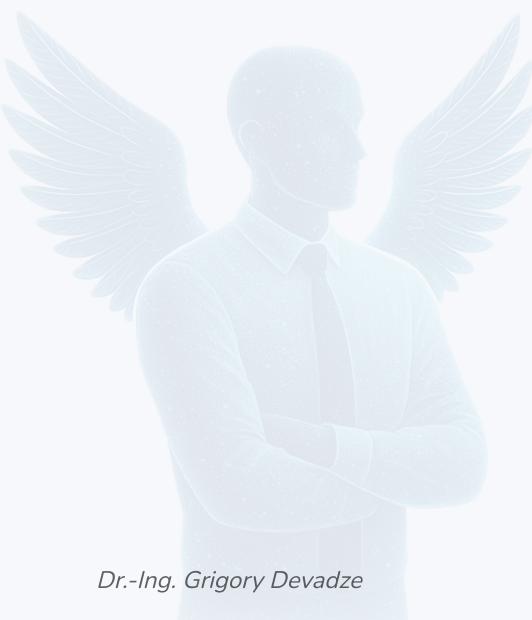
Tangens hyperbolicus (tanh)

- Formel: $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Ausgabe zwischen -1 und 1
- Teils in Hidden-Schichten, GAN und LSTM genutzt



Softmax

- Formel: $f_k(x) = \frac{e^{x_k}}{\sum_{j=1}^K e^{x_j}}$
- Wandelt einen K -dimensionalen Logit-Vektor in Wahrscheinlichkeiten um (Summe = 1)
- In der Ausgabeschicht für Multiklassen-Klassifikation



Was ist One-Hot-Encoding?

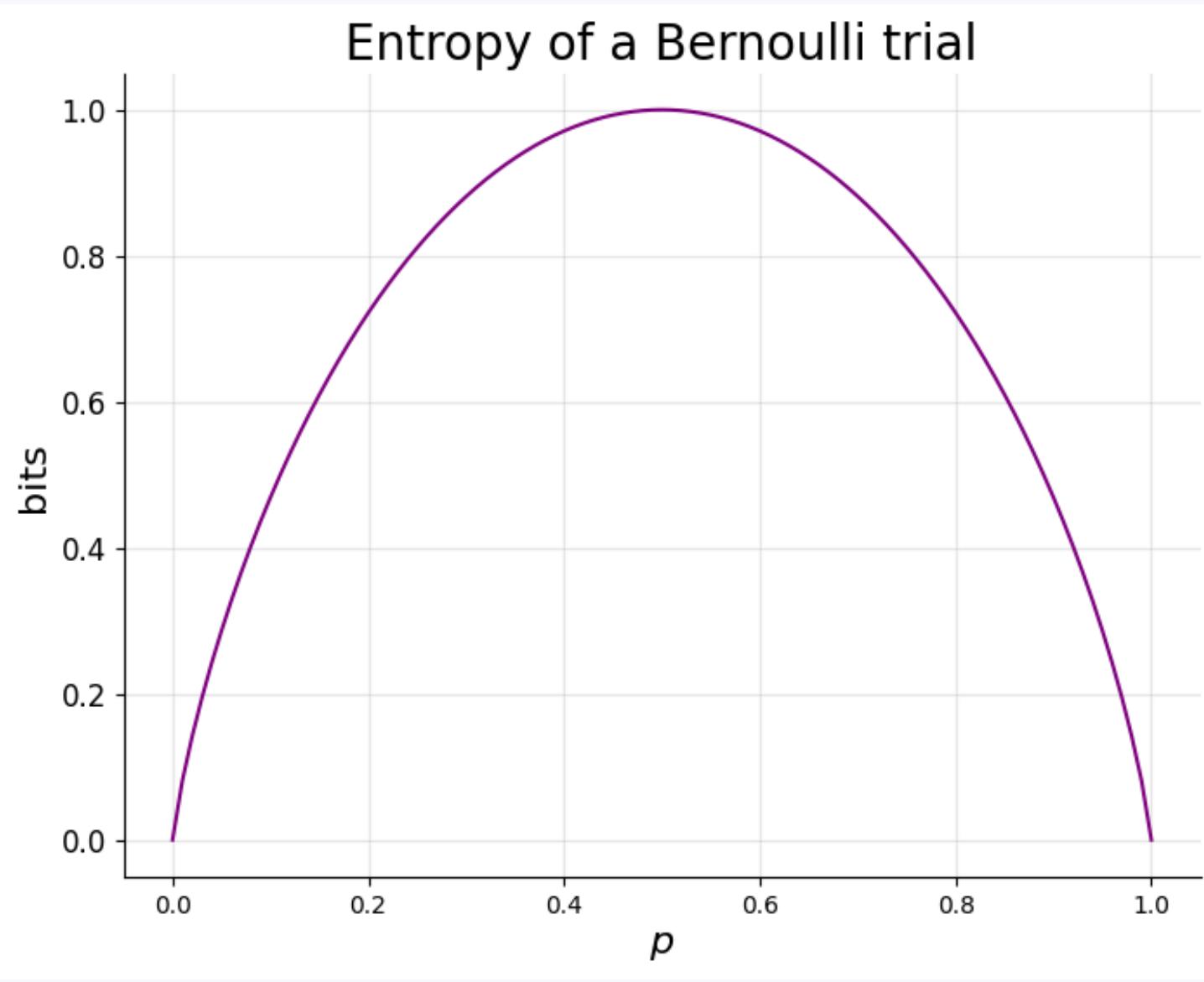
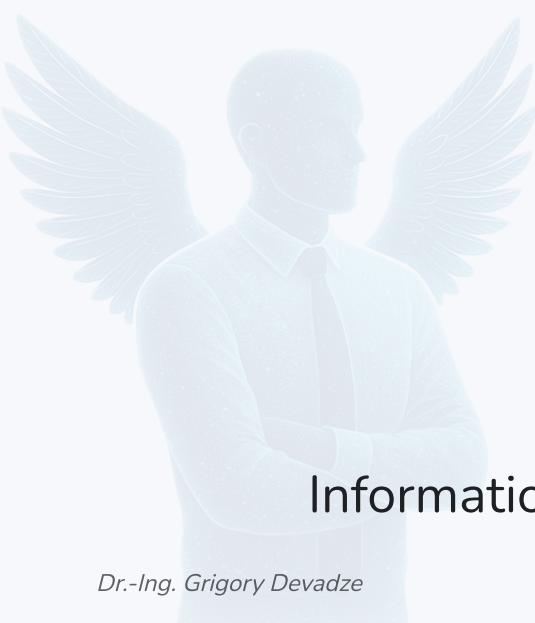
- **One-Hot-Encoding** repräsentiert Kategorien/Klassen als Vektoren:
 - Für K Klassen ist jede Klasse ein K -Vektor mit einer **1** und sonst **0**.
 - Beispiel (3 Klassen):
 - Klasse 1: [1, 0, 0]
 - Klasse 2: [0, 1, 0]
 - Klasse 3: [0, 0, 1]

Binary Cross-Entropy (BCE)

- Für **binäre Klassifikation** (Labels 0/1)
- Vergleicht prognostizierte Wahrscheinlichkeit (\hat{y}) mit dem wahren Label (y):

$$\text{BCE}(y, \hat{y}) = - [y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

- Kleine Loss bei sicheren/korrekteten Vorhersagen; hohe Loss bei falschen/sicheren Vorhersagen



Informationsentropie

Categorical Cross-Entropy (CE)

- Für **Multiklassen-Klassifikation** (Label ist eine Klasse aus K)
- Mit One-Hot-Label y und Prognosen \hat{y} :

$$\text{CE}(y, \hat{y}) = - \sum_{k=1}^K y_k \log(\hat{y}_k)$$

- Bestraft niedrige vorhergesagte Wahrscheinlichkeit für die wahre Klasse

Wann welche Loss?

- **BCE**: Binäre Ausgabe (0/1), Sigmoid in der letzten Schicht
- **CE**: Mehrere Klassen ($1, \dots, K$), Softmax in der letzten Schicht
- **LLMs**: Standard ist Categorical Cross Entropy (CE) auf das nächste Token

Dropout

- Dropout ist eine Regularisierungstechnik für neuronale Netze
- Im Training werden zufällig Anteile der Neuronen (Aktivierungen) auf 0 gesetzt (“ausgeblendet”)
- Reduziert Overfitting, da das Netz sich nicht auf einzelne Knoten „verlassen“ kann
- Dropout-Rate $p \in [0, 1]$: Wahrscheinlichkeit, mit der ein Neuron deaktiviert wird (z.B. $p = 0.5$)

- Im Inferenzmodus (Testen) ist Dropout deaktiviert und die Ausgaben werden skaliert (Weight Scaling)
- In PyTorch: `nn.Dropout(p=0.5)`



Batch-Normalisierung:

- **BatchNorm** normalisiert die Aktivierungen für jedes Mini-Batch
- Zentriert die Eingaben jeder Schicht auf Mittelwert 0 und Varianz 1 und skaliert sie anschließend mit lernbaren Parametern



Wie funktioniert BatchNorm?

- Für jede Batch-Aktivierung x :

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

- γ, β werden gelernt; ϵ verhindert Division durch Null

Warum BatchNorm verwenden?

- Macht das Training schneller und robuster
- Ermöglicht höhere Lernraten und bessere Initialisierung

