

Einführung in Deep Learning

Dr. Konrad Völkel

19. Oktober 2026

1 Überblick

1.1 Über dieses Skript (Voraussetzungen, Lernziele, Literatur)

1.1.1 Einführung in Deep Learning

Über dieses Skript

Dieses Skript begleitet die Vorlesung „Einführung in Deep Learning“ im Bachelor Informatik an der HHU Düsseldorf. Es ist nach Themen gegliedert, nicht nach Vorlesungswochen; Organisatorisches zum jeweiligen Semester (Termine, Räume, Prüfungsmodalitäten) steht nicht hier, sondern wird gesondert bekannt gegeben.

Voraussetzungen

Formale Voraussetzungen: keine.

Inhaltliche Voraussetzungen: Inhalte des Moduls **Data Science**, denn wir setzen rudimentäre Python-Kenntnisse und Grundlagen des maschinellen Lernens voraus, insbesondere die vektorisierte Programmierung mit NumPy und die Idee der linearen Regression. Wo wir auf Stochastik aus dem Modul Data Science zurückgreifen, verlinken wir die passende Stelle im [Data-Science-Skript](#); insbesondere Erwartungswert, Entropie und Maximum Likelihood sollten Sie dort nachschlagen können.

Was wir nicht machen werden

Transformer-Modelle trainieren (GPT usw.), auf großen echten Daten arbeiten, aktuelle Architekturen nachimplementieren, Optimierungstheorie, CUDA-Programmierung, Reinforcement Learning.

Lernziele (Modulhandbuch)

Nach erfolgreicher Teilnahme können die Studierenden:

- Grundbegriffe und -konzepte neuronaler Netze benennen und erklären,
- mathematische Grundlagen für neuronale Netze anwenden,
- einfache Modelle selbst implementieren,
- vortrainierte Modelle in andere Systeme integrieren und
- beurteilen, welche der besprochenen Modelle hinsichtlich einer Anwendung in Frage kommen.

Relation zu anderen Modulen

- Wenig bis keine Überschneidung: Machine Learning, Data Science 2, Introduction to NLP, Python for NLP
- Vertiefung: Deep Learning im Master, zahlreiche darauf aufbauende Kurse wie etwa Advanced Deep Learning, Implementing Transformer Models, Representations in Natural Language Processing, Probabilistic Machine Learning, Reinforcement Learning

Klausurvorbereitung

Das [Glossar](#) listet die Begriffe, die in Klausuraufgaben vorkommen, mit Link zum jeweiligen Kapitel.

Literatur

- Simon J.D. Prince, *Understanding Deep Learning*, MIT Press 2023, frei verfügbar unter <https://udlbook.github.io/udlbook/>
- Deisenroth, Faisal, Ong, *Mathematics for Machine Learning*, Cambridge University Press 2020, frei verfügbar unter <https://mml-book.github.io/>

2 Grundlagen

2.1 Was ist Deep Learning?

2.1.1 Was ist eigentlich Deep Learning?

Der Name ist erfreulich ehrlich: *Deep* heißt „viele Ebenen (Layer)“, *Learning* heißt „Training“. Deep Learning ist also das Training von künstlichen neuronalen Netzen mit vielen Ebenen. Was ein künstliches neuronales Netz ist, was eine Ebene ist und was beim Training passiert, ist der Inhalt dieser Vorlesung. Am Ende des Kapitels haben wir zumindest eine erste Definition.

Was Sie mitbringen sollten

Wir setzen voraus, dass Sie die folgenden Begriffe schon einmal gesehen haben und keine Angst vor ihnen haben: Vektorraum und Vektor, Matrix, Ableitung, Zufallsvariable, und eine grobe Idee davon, was lineare Regression ist. Dazu Grundlagen in Python, idealerweise mit NumPy. Am besten hat man vorher das Modul „Data Science“ bestanden; die Module „Machine Learning“ oder „Data Science 2“ sind *nicht* erforderlich.

Was wir zusammen lernen

Begriffe wie Gradientenabstieg, Loss-Funktion, Matrizen- und Tensorkalkül, latenter Raum, logistische Regression. Die Grundlagen von PyTorch. Was ein neuronales Netz ist, wie man fertige Modelle benutzt, wie man fertige Modelle verändert, und was die Besonderheiten bei Text und bei Bildern sind.

Der grobe Plan in Stichworten: Geschichte (Perzeptron) – Hugging Face (fertige Modelle) – PyTorch (eigene Modelle) – logistische Regression (Softmax) – Gradientenabstieg und Backpropagation – Sparse Autoencoder und VAE für die Klassifikation von MNIST-Ziffern – NLP mit word2vec und Tokenizern – CNNs für MNIST – LLMs benutzen – GANs (wie DALL-E mini).

2.1.2 Wieso Python eine Katastrophe ist

Für maschinelles Lernen ist Python eigentlich völlig ungeeignet, denn es ist

- **single-threaded by default**, mit Garbage Collection und GIL (global interpreter lock), was Multithreading eher erschwert als erleichtert;
- **interpretiert**, nicht direkt auf die Maschine kompiliert, und daher etwas langsam;
- **dynamisch typisiert**, d.h. Typen sind erst zur Laufzeit bekannt, daher aufwändiges Typechecking;
- **nicht maschinennah**, daher ist es schwierig, SIMD-Register und Cachegrößen der spezifischen Hardware zur Optimierung zu nutzen.

2.1.3 Wieso Python ganz toll ist

Für maschinelles Lernen ist Python perfekt, weil

- **Rapid Prototyping** im schnelllebigen ML-Geschäft erforderlich ist;
- die schwierigen Rechnungen in **C, Rust oder CUDA** durchgeführt werden und Python nur die Funktionsaufrufe macht;
- die Sprache sehr **populär** ist und damit viele gut gewartete Pakete das Datenhandling sehr leicht machen;
- man damit alle **Prinzipien** schnell lernen kann, die man später immer noch in anderen Sprachen implementieren könnte;
- damit trainierte Modelle (die **Gewichte**) unabhängig von Python zur Inferenz verwendet werden können.

Der scheinbare Widerspruch löst sich auf, wenn man Python als *Steuersprache* betrachtet: Die eigentliche Arbeit – Matrixmultiplikationen auf Millionen von Zahlen – passiert in kompiliertem Code, oft auf der GPU. Python sagt nur, welche Operation als Nächstes dran ist. Genau das ist die Idee der vektorisierten Programmierung mit NumPy, und PyTorch treibt sie weiter.

Lesetext zum Einstieg

How AMD May Get Across the CUDA Moat (HPCwire, 2023). Fragen zum Text:

1. Was ist der Unterschied zwischen GPU und SIMD?
2. Wer entwickelt PyTorch?
3. Was ist der Unterschied zwischen CUDA und ROCm?

Antworten zum Lesetext

1. **GPU und SIMD.** SIMD (*single instruction, multiple data*) ist ein Rechenprinzip: dieselbe Operation gleichzeitig auf vielen Datenelementen. Eine GPU ist ein Produkt, das dieses Prinzip mit tausenden kleinen Rechenkernen umsetzt, ursprünglich für die parallele Berechnung von Bildpunkten. Für Deep Learning zählt nur das Prinzip, denn Matrixmultiplikation ist SIMD-Arbeit. Deshalb eignet sich eine Gaming-Grafikkarte zum Trainieren, und deshalb sind die Chips in Rechenzentren, die nie ein Bild ausgeben, streng genommen keine *Graphics Processing Units* mehr.
2. **PyTorch** wurde bei Facebook AI Research entwickelt (heute Meta) und wird seit 2022 unter dem Dach der Linux Foundation als PyTorch Foundation weiterentwickelt; Meta bleibt der größte Beitragende. TensorFlow, die Konkurrenz von Google, hat an Bedeutung verloren.
3. **CUDA und ROCm** sind die Software-Plattformen (Compiler, Bibliotheken, Treiber), mit denen man SIMD-Programme auf NVIDIA- bzw. AMD-Hardware schreibt und ausführt. CUDA ist proprietär und seit 2007 der De-facto-Standard, auf dem PyTorch und fast alle Forschungscode aufsetzen. ROCm ist AMDs quelloffenes Gegenstück; die Schicht HIP bildet die CUDA-Schnittstelle nach, aber ROCm kam später und hat weniger fertige Bibliotheken. Genau das ist der *moat* aus dem Titel: nicht die Hardware, sondern die Software drumherum. Wir programmieren in dieser Vorlesung weder CUDA noch ROCm, sondern PyTorch, das beides als Backend benutzen kann.

2.1.4 Erinnerung: Begriffe aus dem maschinellen Lernen

Notation: Datenpunkt, Feature, Stichprobe

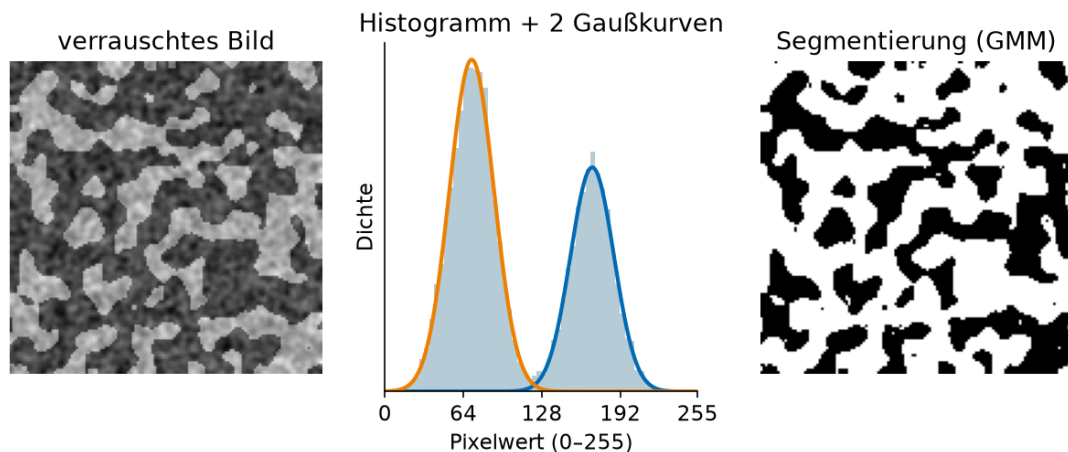
Ein einzelner Datenpunkt ist ein Vektor $x \in \mathbb{R}^n$; seine Komponenten $x_1, \dots, x_n$ heißen *Features*. Eine Stichprobe besteht aus N solchen Vektoren. Wenn wir beides gleichzeitig brauchen, schreiben wir den Datenpunkt-Index hochgestellt in Klammern: $x^{(1)}, \dots, x^{(N)}$, und $x_j^{(i)}$ ist Feature j von Datenpunkt i . In NumPy ist das `x[i, j]`: eine Zeile pro Datenpunkt, eine Spalte pro Feature. In Summen über die Stichprobe, in denen die Features nicht einzeln vorkommen, schreiben wir kürzer x_i für den i -ten Datenpunkt.

Für das Skalarprodukt benutzen wir $w \cdot x$ und $\langle w, x \rangle$ gleichbedeutend, beides ist $\sum_{j=1}^n w_j x_j$.

Daten und erzeugende Verteilung

Wir haben Datenpunkte $x_1, x_2, \dots, x_N$ mit jeweils $x_i \in \mathbb{R}^n$. Die Grundannahme des maschinellen Lernens ist, dass es eine *erzeugende Verteilung* (die Population) einer Zufallsvariable X gibt, aus der die Datenpunkte als Stichprobe gezogen wurden – und die wir nicht kennen.

Ein **Modell** für X ist eine Beschreibung dieser Verteilung, etwa durch eine Dichtefunktion $p(x)$ oder durch einen *Sampler*, also einen Algorithmus, der neue Stichproben $\tilde{x} \sim X$ erzeugt. Ein solches Modell heißt **generativ**, weil man damit Stichproben generieren kann. Ein klassisches Beispiel ist ein Gaussian Mixture Model, mit dem man etwa ein Bild in Vorder- und Hintergrund segmentieren kann.



Ein generatives Modell in Aktion: Ein Gaussian Mixture Model mit zwei Komponenten wird an das Histogramm der Pixelwerte gefittet und trennt so Vorder- und Hintergrund (nach einem Beispiel aus den Scipy Lectures).

Supervised und Unsupervised

Wenn es zu den Datenpunkten auch (bekannte) **Label** gibt, d.h. nicht nur $x_1, \dots, x_N$, sondern auch $y_1, \dots, y_N$ mit $y_i \in \mathbb{R}^m$, so sprechen wir von **Supervised Learning**. Oft ist $m = 1$ (einfache Regression) oder sogar $y_i \in \{0, 1\}$ (binäre Klassifikation). Ohne Label ist es **Unsupervised Learning**; dazu gehören generative Modelle wie oben, aber auch Clustering und Dimensionsreduktion.

In beiden Fällen unterscheiden wir zwei Phasen:

- **Training:** Input sind die Daten, Output ist ein *gefittetes Modell* (bei uns: ein Satz Zahlen, die Gewichte).
- **Inferenz:** Input sind neue Daten und das gefittete Modell, Output ist die Vorhersage.

Regression und Klassifikation

Regression ist supervised, d.h. es gibt (x, y) -Paare. Das Ziel ist ein funktionaler Zusammenhang

$$Y = f(X) + \varepsilon, \quad (1)$$

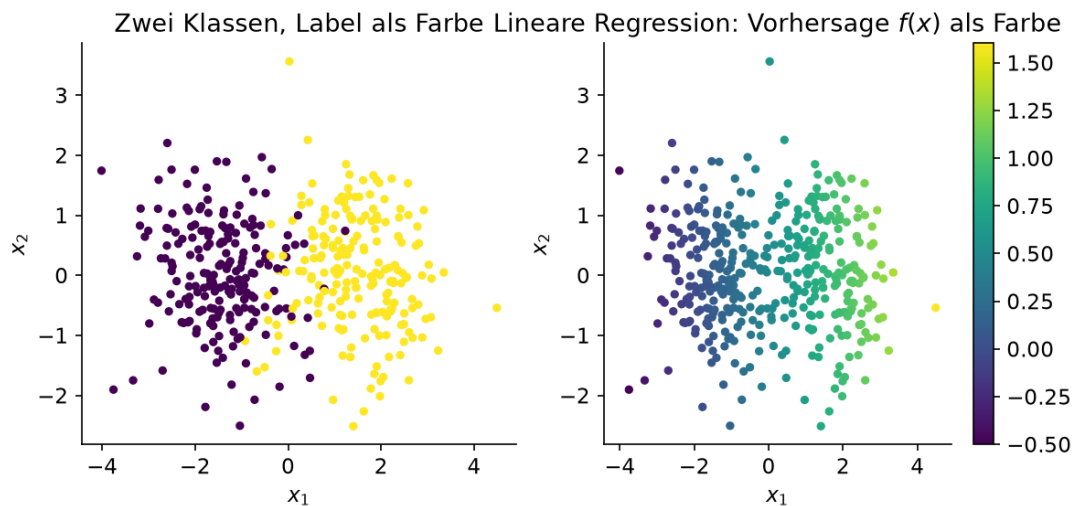
sodass für die Stichprobe (x_i, y_i) der Fehler ε möglichst klein ausfällt. Bei der linearen Regression ist f affin-linear, $f(x) = w \cdot x + b$.

Klassifikation ist der Spezialfall, in dem Y nur endlich viele Werte annimmt, z.B. $Y \in \{0, 1\}$. Man kann Klassifikation als Regression betreiben – man fittet f mit reellen Werten und färbt anschließend nach $f(x) > \frac{1}{2}$ – aber wir werden sehen, dass das nicht die beste Idee ist und die logistische Regression es besser macht.

Die Kernfrage: Wie gut ist das Modell?

Wie bewerte ich die Güte des Modells, wie bewerte ich den Erfolg der Inferenz? Die Standardantwort ist der **Training-Test-Split**: Man hält einen Teil der Datenpunkte zurück (verwendet ihn nicht im Training), um damit das trainierte Modell zu testen. Die kompliziertere Variante ist die k -fold cross validation, bei der man die Daten in k Teile zerlegt und reihum jeden Teil einmal als Testset verwendet.

Und was misst man auf dem Testset? Bei Klassifikation naheliegend die **Accuracy**, den Anteil korrekt klassifizierter Samples. Die ist irreführend, wenn die Klassen unbalanciert sind (99 % Nicht-Spam: ein Modell, das immer „kein Spam“ sagt, hat 99 % Accuracy). Dann schaut man pro Klasse auf **Precision** (welcher Anteil der als positiv vorhergesagten Samples ist wirklich positiv?) und **Recall** (welcher Anteil der wirklich positiven Samples wurde gefunden?); bei Regression auf den mittleren quadratischen Fehler. Wir kommen bei den Loss-Funktionen darauf zurück.



Zwei linear trennbare Klassen (links) und eine lineare Regression mit der Klasse als abhängiger Variable (rechts): Die Vorhersage ist eine reelle Zahl, die man bei $\frac{1}{2}$ abschneiden kann.

Quizfrage

Was kann beim Vergleich verschiedener Modelle und der Optimierung von Hyperparametern mit dem Testset passieren?

Antwort: Wer Modelle vergleicht und Hyperparameter nach dem Testergebnis wählt, lässt das Testset *indirekt* ins Modell einfließen – die gemessene Güte ist dann zu optimistisch. Deshalb spaltet man vom Trainingsset ein **Validation-Set** ab (**Validation-Split**), trifft daran alle Entscheidungen und fasst das Testset genau einmal an, ganz am Ende.

2.1.5 Kurzgeschichte

Bis 2010

- **1943:** McCulloch und Pitts modellieren ein menschliches Hirnneuron mit elektrischen Schaltkreisen.
- **1958:** Rosenblatt entwirft das **Perzeptron** – und es wurde tatsächlich gebaut: Der *Mark I Perceptron* am Cornell Aeronautical Laboratory war ein frühes System zur Mustererkennung, mit Fotozellen als Eingabe und Potentiometern als Gewichten, die von Elektromotoren nachgestellt wurden.
- Das Perzeptron kann nur linear trennbare Klassen unterscheiden lernen (dazu gleich mehr). Diese Einsicht, prominent im Buch *Perceptrons* von Minsky und Papert (1969), gilt als Auslöser des ersten **AI Winter**. Wie viel an dieser offiziellen Geschichte dran ist, untersucht lesenswert: Olazaran, M. (1996). *A Sociological Study of the Official History of the Perceptrons Controversy*. Social Studies of Science, 26(3), 611–659.
- **1986:** Backpropagation wird (wieder-)erfunden bzw. populär, mehr Schichten werden verwendet.

2010–2015

Lesetipp: „The Decade of Deep Learning“ von Leo Gao, bmk.sh/2019/12/31/The-Decade-of-Deep-Learning. Die Highlights für diese Vorlesung:

- **2012** AlexNet – „the paper that sparked the field of Deep Learning“: ein CNN gewinnt die ImageNet-Challenge mit großem Abstand, trainiert auf GPUs.
- **2013** word2vec – Repräsentation von Wörtern als Vektoren in einem latenten Raum.
- **2014** GANs – erstmals „realistische“ Bildgenerierung.

Nach 2015

Das meiste davon ist Stoff der aufbauenden Module. Die Transformer sind die Ausnahme:

- **2017** Transformer, „Attention is all you need“ – die Grundlage für GPT. Die sehen wir uns im Kapitel „[Transformer](#)“ an und benutzen sie im Kapitel „[LLMs benutzen](#)“ – trainieren werden wir aber keine.
- **2020** AlphaFold 2 „löst“ das Proteinfaltungsproblem – Nobelpreis für Chemie 2024.
- **2021** Diffusionsmodelle schlagen GANs bei der Bildgenerierung (Dhariwal und Nichol, *Diffusion Models Beat GANs on Image Synthesis*); das Verfahren selbst, DDPM, ist von 2020.
- **2022** Veröffentlichung von ChatGPT; danach RLHF, Copilot, Multimodalität, RAG, Agents, ...
- **2024/25** *Agentic Coding*: Modelle bearbeiten echte GitHub-Issues selbstständig – Code lesen, ändern, Tests laufen lassen, nachbessern. Gemessen wird das auf **SWE-bench Verified**, 500 von Menschen geprüften Aufgaben aus zwölf Python-Projekten; bestanden ist eine Aufgabe erst, wenn die Tests des Projekts durchlaufen. Innerhalb von zwei Jahren ging das von wenigen Prozent auf über die Hälfte.

Diese Prozentzahlen sind die unzuverlässigsten in diesem Kapitel, und zwar aus einem Grund, der über den Benchmark hinausgeht: **Ein Modell wird hier nie allein gemessen, sondern immer zusammen mit seinem Harness.** Das Harness ist das Programm drumherum – es entscheidet, welche Dateien das Modell zu sehen bekommt, welche Werkzeuge es aufrufen darf, was passiert, wenn ein Test fehlschlägt und was aus dem Verlauf weggeworfen wird, wenn der Kontext voll ist. Lewis (2026, *Same Model, Different Harness: Different Coding-Agent Results*) hält die Gewichte fest und ändert nur, wie das Harness ältere Werkzeugausgaben kürzt: Der Anteil reparierter Tests steigt von 28 % auf 49 %. Einundzwanzig Prozentpunkte, ohne dass am Modell etwas geändert wurde.

Daraus folgen zwei Dinge. Erstens ist die Frage „welches Modell ist besser?“ in dieser Disziplin schlecht gestellt – gemessen wird das Paar aus Modell und Harness. Zweitens sind Zahlen aus verschiedenen Quellen nur dann vergleichbar, wenn dasselbe Harness benutzt wurde; selbst der Benchmark-Betreiber Epoch AI hat sein Standard-Scaffold im Februar 2026 ausgetauscht, worauf die Werte aller Modelle sprangen. Wer eine solche Zahl zitiert, nennt das Harness dazu – sonst ist sie ungefähr so aussagekräftig wie eine Laufzeit ohne Angabe des Rechners.

2.1.6 Übung

Ein Rissdetektor findet unter 200 Bauteilen 30 als „rissig“ markierte, davon sind 24 tatsächlich rissig; insgesamt gibt es 40 rissige Bauteile. Berechnen Sie Precision und Recall. Welche der beiden Zahlen würden Sie in der Qualitätssicherung höher gewichten, und warum?

Antwortprobe: die beiden Werte verhalten sich wie 4 zu 3, und der größere ist der, in dessen Nenner die 30 steht.

Weiterlesen

Prince, *UDL*, Kap. 1 (Introduction) und 2 (Supervised learning); Deisenroth et al., *MLL*, Kap. 8 (When models meet data). Beide Bücher sind frei verfügbar, siehe [Über dieses Skript](#).

2.2 Hintergrundwissen Lineare Algebra

Im Inneren eines Neurons steckt ein Skalarprodukt, in einer Schicht eine Matrix, in der Backpropagation ein Matrixprodukt. Dieses Kapitel stellt die Begriffe bereit, auf denen der Rest des Buchs aufbaut, und legt die Schreibweisen fest.

Ein Nachschlagkapitel, kein Lehrgang: Wer aus der linearen Algebra kommt, überfliegt es. Die Eigenschaften, die unten ohne Beweis stehen, sind in zwei, drei Zeilen nachzurechnen.

2.2.1 Vektoren

Definition 2.1 (Vektor). *Ein reeller Vektor der Dimension $d \in \mathbb{N}$ ist ein Tupel von d reellen Zahlen, den Komponenten $v_i \in \mathbb{R}$, geschrieben als Spalte:*

$$v = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix} \in \mathbb{R}^d. \quad (2)$$

Es ist $v = w$, wenn $v_i = w_i$ für alle i gilt.

Das hochgestellte $\top$ ist das *Transponieren*: Es vertauscht Zeilen und Spalten, aus der Spalte v wird die Zeile $v^\top = (v_1, \dots, v_d)$. Die *Standardbasisvektoren* $e_1, \dots, e_d$ haben eine 1 in einer Komponente und sonst Nullen.

Spalte, Zeile und was NumPy daraus macht

`np.array([1.0, 2.0])` hat die Shape `(2,)` – weder Zeile noch Spalte, und `v.T` tut nichts. Erst `v.reshape(2, 1)` ist die Spalte. Das ist der häufigste Grund dafür, dass eine Rechnung auf dem Papier stimmt und im Code eine Shape zu viel hat; siehe [Kapitel „Tensoren“](#).

2.2.2 Hadamard-Produkt und Skalarprodukt

Definition 2.2 (Hadamard-Produkt). *Das Hadamard-Produkt von $v, w \in \mathbb{R}^d$ ist der Vektor $v \odot w \in \mathbb{R}^d$ mit $(v \odot w)_i := v_i w_i$, also die komponentenweise Multiplikation.*

Definition 2.3 (Skalarprodukt). *Das Skalarprodukt von $v, w \in \mathbb{R}^d$ ist die Zahl*

$$v \cdot w := \sum_{i=1}^d (v \odot w)_i = \sum_{i=1}^d v_i w_i \in \mathbb{R}. \quad (3)$$

Wie im [Kapitel „Was ist Deep Learning?“](#) festgelegt, sind $v \cdot w$ und $\langle v, w \rangle$ dasselbe; unten kommt mit $v^\top w$ eine dritte Schreibweise dazu.

Example 2.1 (Skalarprodukt). *Für $v = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$ und $w = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$ ist $v \odot w = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$ und $v \cdot w = 2 \cdot 1 + 0 \cdot 3 = 2$. Mit den Standardbasisvektoren liest man Komponenten aus: $e_1 \cdot \begin{pmatrix} x \\ y \end{pmatrix} = x$.*

Drei Eigenschaften benutzt das Buch, ohne sie jedes Mal zu nennen. Das Skalarprodukt ist

- **symmetrisch**: $v \cdot w = w \cdot v$,
- **bilinear**: $(\lambda u + \mu v) \cdot w = \lambda(u \cdot w) + \mu(v \cdot w)$, und ebenso im zweiten Argument,
- **positiv definit**: $v \cdot v \geq 0$, mit $v \cdot v = 0$ nur für den Nullvektor.

Die Bilinearität trägt den Konvergenzbeweis im [Kapitel „Das Perzeptron trainieren“](#), die positive Definitheit den nächsten Abschnitt.

2.2.3 Länge und Normalisierung

Definition 2.4 (Länge). Die Länge (2-Norm) von $v \in \mathbb{R}^d$ ist $\|v\| := \sqrt{v \cdot v} = \sqrt{v_1^2 + \dots + v_d^2}$.

Dass unter der Wurzel nichts Negatives steht, ist die positive Definitheit; in $\mathbb{R}^2$ ist es der Satz des Pythagoras. Es gilt $\|v\| = 0$ nur für den Nullvektor, und $\|\lambda v\| = |\lambda| \|v\|$ – mit Betrag, denn eine Länge ist nie negativ.

Example 2.2 (Länge). $\left\| \begin{pmatrix} 2 \\ 0 \end{pmatrix} \right\| = 2$ und $\left\| \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right\| = \sqrt{9 + 16} = 5$. Standardbasisvektoren haben Länge 1.

Definition 2.5 (Normalisierung). Für $v \neq 0$ ist $\tilde{v} := v/\|v\|$ die Normalisierung von v ; sie hat Länge 1. Vektoren der Länge 1 heißen Einheitsvektoren.

Damit zerfällt jeder Vektor in Richtung und Länge, $v = \|v\| \tilde{v}$, und mit ihm das Skalarprodukt:

$$v \cdot w = \|v\| \|w\| (\tilde{v} \cdot \tilde{w}). \quad (4)$$

Der zweite Faktor hängt nur von den Richtungen ab und bekommt im nächsten Abschnitt einen Namen. Nebenbei: Aus $1 = v_1^2 + \dots + v_d^2$ folgt $-1 \leq v_i \leq 1$, normalisierte Embeddings liefern also gutmütige Zahlen.

Obacht: Wurzeln kosten, Nullen stören

Zum Vergleichen zweier Längen braucht man keine Wurzel: $\|v\| < \|w\|$ gilt genau dann, wenn $v \cdot v < w \cdot w$. Und `v / np.linalg.norm(v)` ist für $v = 0$ eine Division durch null – in echten Daten kommt der Nullvektor häufiger vor, als man denkt.

2.2.4 Winkel

Definition 2.6 (Winkel). Für $v, w \neq 0$ ist der Winkel $\theta \in [0, \pi]$ zwischen ihnen festgelegt durch

$$\cos \theta = \frac{v \cdot w}{\|v\| \|w\|} = \tilde{v} \cdot \tilde{w}. \quad (5)$$

Dass der Bruch zwischen -1 und 1 liegt, also überhaupt ein Kosinus sein kann, ist die Ungleichung von Cauchy und Schwarz: $|v \cdot w| \leq \|v\| \|w\|$.

Example 2.3 (Winkel). Für $v = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$ und $w = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$ ist $v \cdot w = 2$, $\|v\| = 2$ und $\|w\| = \sqrt{10} \approx 3,162$, also $\cos \theta = 1/\sqrt{10} \approx 0,316$ und $\theta \approx 71,6^\circ$.

Der Kosinus misst Ähnlichkeit der Richtungen, unabhängig von den Längen. Als **Kosinusähnlichkeit** ist er das Standardmaß für Embeddings, etwa im [Kapitel „word2vec“](#) und im [Kapitel „Latente Räume“](#).

2.2.5 Orthogonalität

Definition 2.7 (Orthogonalität). $v, w \in \mathbb{R}^d$ heißen orthogonal, geschrieben $v \perp w$, wenn $v \cdot w = 0$ ist.

Für Vektoren ungleich null ist das der Fall $\cos \theta = 0$, also ein rechter Winkel; die Standardbasisvektoren sind paarweise orthogonal. Daran hängt die Geometrie des Neurons: Der Gewichtsvektor steht senkrecht auf der Trennebene, und die Aktivierung misst den Abstand zu ihr. Ausgerechnet wird das im [Kapitel „Neuron und Perzeptron“](#).

2.2.6 Matrizen

Definition 2.8 (Matrix). Eine reelle Matrix mit m Zeilen und n Spalten ist ein Schema $A = (a_{ij}) \in \mathbb{R}^{m \times n}$ mit Einträgen $a_{ij} \in \mathbb{R}$; der erste Index zählt die Zeilen, der zweite die Spalten. Ein Vektor aus $\mathbb{R}^d$ ist der Spezialfall $\mathbb{R}^{d \times 1}$.

Die Angabe $m \times n$ heißt im Code *Shape*. Sie mitzuführen ist die halbe Miete: Die meisten Fehler beim Rechnen mit Netzen sind Shape-Fehler.

Definition 2.9 (Matrix mal Vektor). Für $A \in \mathbb{R}^{m \times n}$ und $x \in \mathbb{R}^n$ ist $Ax \in \mathbb{R}^m$ mit $(Ax)_i = \sum_{j=1}^n a_{ij} x_j$: Die i -te Komponente ist das Skalarprodukt der i -ten Zeile von A mit x .

Example 2.4 (Matrix mal Vektor). $A = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 3 \end{pmatrix} \in \mathbb{R}^{2 \times 3}$ und $x = (2, 1, 4)^\top$ ergeben $Ax = \begin{pmatrix} 2+2+0 \\ 0+1+12 \end{pmatrix} = \begin{pmatrix} 4 \\ 13 \end{pmatrix} \in \mathbb{R}^2$.

So liest man eine Schicht: In $Wx + b$ mit $W \in \mathbb{R}^{m \times n}$ steht in jeder Zeile der Gewichtsvektor eines Neurons, und $Wx + b$ berechnet m Aktivierungen auf einmal. Eine Matrix *ist* eine lineare Abbildung $\mathbb{R}^n \rightarrow \mathbb{R}^m$.

Definition 2.10 (Matrix mal Matrix). Für $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{n \times k}$ ist $AB \in \mathbb{R}^{m \times k}$ mit $(AB)_{i\ell} = \sum_{j=1}^n a_{ij}b_{j\ell}$.

Die Shape-Regel steht in der Definition: $(m \times n) \cdot (n \times k)$ ergibt $m \times k$, die inneren Zahlen müssen übereinstimmen und fallen weg. Passen sie nicht, ist das Produkt nicht definiert; AB und BA sind verschieden, oft ist nur eines von beiden definiert. Inhaltlich ist das Produkt die **Verkettung** der Abbildungen: erst B , dann A . Daran hängt die Backpropagation, denn die Ableitung einer Verkettung ist das Produkt der Ableitungen ([Kapitel „Gradientenabstieg“](#)).

Definition 2.11 (Transponieren). Die Transponierte von $A \in \mathbb{R}^{m \times n}$ ist $A^\top \in \mathbb{R}^{n \times m}$ mit $(A^\top)_{ij} = a_{ji}$: Zeilen werden Spalten.

Drei Regeln, die das Buch braucht: $(A^\top)^\top = A$; $(AB)^\top = B^\top A^\top$, die Reihenfolge dreht sich also um; und $v \cdot w = v^\top w$, denn eine Zeile mal eine Spalte ist eine 1×1 -Matrix. Die zweite Regel ist der Grund, warum in der Backpropagation ständig transponierte Gewichtsmatrizen auftauchen: vorwärts W , rückwärts $W^\top$.

Example 2.5 (Produkt und Transponierte). Mit A wie oben und $B = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 1 & 3 \end{pmatrix} \in \mathbb{R}^{3 \times 2}$ ist

$$AB = \begin{pmatrix} 5 & 2 \\ 5 & 10 \end{pmatrix} \in \mathbb{R}^{2 \times 2}, \quad (AB)^\top = \begin{pmatrix} 5 & 5 \\ 2 & 10 \end{pmatrix} = B^\top A^\top. \quad (6)$$

BA ist ebenfalls definiert, aber 3×3 – schon die Shape zeigt, dass die Reihenfolge zählt.

Zeilen statt Spalten: die Konvention im Code

Auf dem Papier ist ein Datenpunkt eine Spalte, und eine Schicht rechnet Wx . In NumPy und PyTorch liegt ein Batch als Matrix der Shape (N, n) vor, ein Datenpunkt je *Zeile*, und dieselbe Schicht rechnet $X @ W.T + b$. Beide Konventionen beschreiben dasselbe; transponiert ist genau der Unterschied. Wer eine Formel aus dem Buch in Code übersetzt und eine Shape-Fehlermeldung bekommt, hat meist diesen Wechsel vor sich und keinen Rechenfehler.

Weiterlesen

Deisenroth et al., *MML*, Kap. 2 (Linear algebra) und Kap. 3 (Analytic geometry: Skalarprodukt, Winkel, Projektion). Frei verfügbar, siehe [Über dieses Skript](#).

2.3 Hintergrundwissen Analysis

Training heißt ableiten: Der Gradient sagt, wohin ein Schritt gehen soll, und die Kettenregel verteilt ihn über die Schichten. Dieses Kapitel stellt die Begriffe bereit, die dafür gebraucht werden, von der partiellen Ableitung bis zur Jacobi-Matrix, und erklärt, was man tut, wenn eine Funktion einen Knick hat.

Wie das [Kapitel zur linearen Algebra](#) ein Nachschlagkapitel. Wie der Gradient *benutzt* wird, steht im [Kapitel „Gradientenabstieg“](#); hier steht, was er ist.

2.3.1 Partielle Ableitung

Definition 2.12 (Partielle Ableitung). Sei $f: \mathbb{R}^n \rightarrow \mathbb{R}$ und $a \in \mathbb{R}^n$. Die partielle Ableitung von f nach der j -ten Variablen an der Stelle a ist

$$\frac{\partial f}{\partial x_j}(a) := \lim_{h \rightarrow 0} \frac{f(a + h e_j) - f(a)}{h}, \quad (7)$$

sofern der Grenzwert existiert. Man hält also alle Komponenten bis auf die j -te fest und leitet die verbleibende Funktion einer Variablen ganz gewöhnlich ab.

Example 2.6 (Partielle Ableitungen). Für $f(x_1, x_2) = x_1^2 + 3x_1x_2$ ist $\frac{\partial f}{\partial x_1} = 2x_1 + 3x_2$ und $\frac{\partial f}{\partial x_2} = 3x_1$. An der Stelle $a = (1, 2)^\top$ sind das 8 und 3.

2.3.2 Richtungsableitung und Gradient

Die partiellen Ableitungen messen die Änderung entlang der Koordinatenachsen. Eine beliebige Richtung geht genauso.

Definition 2.13 (Richtungsableitung). Für einen Einheitsvektor $u \in \mathbb{R}^n$ ist die Richtungsableitung von f in Richtung u an der Stelle a

$$D_u f(a) := \lim_{h \rightarrow 0} \frac{f(a + hu) - f(a)}{h}. \quad (8)$$

Für $u = e_j$ ist das die j -te partielle Ableitung.

Definition 2.14 (Gradient). Der Gradient von $f: \mathbb{R}^n \rightarrow \mathbb{R}$ an der Stelle a ist der Vektor der partiellen Ableitungen,

$$\nabla f(a) := \left(\frac{\partial f}{\partial x_1}(a), \dots, \frac{\partial f}{\partial x_n}(a) \right)^\top \in \mathbb{R}^n. \quad (9)$$

Beide Begriffe hängen zusammen: Ist f differenzierbar, so gilt $D_u f(a) = \nabla f(a) \cdot u$. Die Richtungsableitung ist also ein Skalarprodukt, und damit ist sofort klar, welche Richtung die steilste ist. Nach Cauchy-Schwarz ist $|D_u f(a)| \leq \|\nabla f(a)\|$, mit Gleichheit genau für $u = \nabla f(a) / \|\nabla f(a)\|$:

Der Gradient zeigt in die Richtung des steilsten Anstiegs, und seine Länge ist die Steigung in dieser Richtung.

Das ist die Rechtfertigung für den Gradientenabstieg: Wer absteigen will, geht entgegen dem Gradienten.

Example 2.7 (Steilste Richtung). Für f wie oben ist $\nabla f(1, 2) = (8, 3)^\top$. In Richtung e_1 steigt f mit 8, in Richtung e_2 mit 3, am steilsten in Richtung $(8, 3)^\top / \sqrt{73}$ mit der Steigung $\sqrt{73} \approx 8,544$.

2.3.3 Die Ableitung als beste lineare Näherung

Für Funktionen zwischen Vektorräumen ist die Ableitung keine Zahl mehr, sondern eine lineare Abbildung, also eine Matrix.

Definition 2.15 (Ableitung und Jacobi-Matrix). $g: \mathbb{R}^n \rightarrow \mathbb{R}^m$ heißt differenzierbar in a , wenn es eine lineare Abbildung $(Dg)(a): \mathbb{R}^n \rightarrow \mathbb{R}^m$ gibt mit

$$g(a + h) = g(a) + (Dg)(a)h + r(h), \quad \frac{\|r(h)\|}{\|h\|} \rightarrow 0 \text{ für } h \rightarrow 0. \quad (10)$$

Die zugehörige Matrix heißt *Jacobi-Matrix*; ihre Einträge sind die partiellen Ableitungen der Komponentenfunktionen,

$$((Dg)(a))_{ij} = \frac{\partial g_i}{\partial x_j}(a) \quad (i = 1, \dots, m, j = 1, \dots, n), \quad (11)$$

sie hat also m Zeilen und n Spalten.

Der Rest r verschwindet schneller als h selbst: Die Jacobi-Matrix ist die *beste* lineare Näherung an g in der Nähe von a . Für $m = 1$ ist sie die Zeile $\nabla f(a)^\top$ – Gradient und Ableitung sind dieselbe Information, einmal als Spalte und einmal als Zeile geschrieben.

Example 2.8 (Jacobi-Matrix). Für $g(x_1, x_2) = (x_1 x_2, x_1 + x_2^2)^\top$ ist

$$(Dg)(x) = \begin{pmatrix} x_2 & x_1 \\ 1 & 2x_2 \end{pmatrix}, \quad (Dg)(1, 2) = \begin{pmatrix} 2 & 1 \\ 1 & 4 \end{pmatrix}. \quad (12)$$

Die Kettenregel in dieser Sprache – die Ableitung einer Verkettung ist das Matrixprodukt der Ableitungen – steht im [Kapitel „Gradientenabstieg“](#) und ist der Kern der Backpropagation.

2.3.4 Zweite Ableitungen: die Hesse-Matrix

Definition 2.16 (Hesse-Matrix). Für $f: \mathbb{R}^n \rightarrow \mathbb{R}$ mit zweiten partiellen Ableitungen ist die *Hesse-Matrix* an der Stelle a

$$H_f(a) := \left(\frac{\partial^2 f}{\partial x_i \partial x_j}(a) \right)_{i,j} \in \mathbb{R}^{n \times n}. \quad (13)$$

Sind die zweiten Ableitungen stetig, ist H_f symmetrisch (Satz von Schwarz). Sie beschreibt die Krümmung. Der Gradient sagt nur, in welche Richtung es abwärts geht; die Hesse-Matrix sagt zusätzlich, wie stark sich diese Richtung ändert, und unterscheidet an einem kritischen Punkt Minimum, Maximum und Sattelpunkt.

Im Deep Learning kommt sie trotzdem kaum vor, und der Grund ist eine Abschätzung: n ist hier die Zahl der Parameter. Schon ein kleines Netz mit einer Million Parametern hätte eine Hesse-Matrix mit 10^{12} Einträgen, in `float32` also vier Terabyte. Deshalb arbeiten die Optimierer dieses Buchs mit Verfahren *erster* Ordnung und schätzen Krümmung höchstens billig und implizit, über Momentum und adaptive Lernraten ([Kapitel „Optimierer“](#)).

2.3.5 Schwache Ableitung

ReLU hat bei 0 einen Knick, und dort existiert die Ableitung nicht: Die einseitigen Grenzwerte des Differenzenquotienten sind verschieden, es gibt keine Tangente. Dasselbe gilt für den Betrag $|x|$. Trainiert wird trotzdem, und der Begriff, der das rechtfertigt, ist der folgende.

Definition 2.17 (Schwache Ableitung). $g: \mathbb{R} \rightarrow \mathbb{R}$ heißt *schwache Ableitung* von $f: \mathbb{R} \rightarrow \mathbb{R}$, wenn für alle $a < b$ gilt

$$\int_a^b g(x) dx = f(b) - f(a). \quad (14)$$

Verlangt wird also nur die Eigenschaft, die eine gewöhnliche Ableitung im Hauptsatz hat. Wo f differenzierbar ist, stimmt jede schwache Ableitung mit f' überein; an einzelnen Stellen dagegen ist sie **nicht eindeutig**, denn ein Integral ändert sich nicht, wenn man den Wert an endlich vielen Stellen ändert.

Example 2.9 (Der Betrag). $f(x) = |x|$ hat als schwache Ableitung die Vorzeichenfunktion sign mit $\text{sign}(0) = 0$. Dass gerade 0 dort steht, ist Konvention, nicht Rechnung: Jeder andere Wert an dieser einen Stelle täte es auch.

Genau diese Freiheit nutzt ein Framework bei ReLU: Es gibt die Ableitung auf beiden Seiten des Knicks zurück und legt sich an der Knickstelle auf einen Wert fest. Dass die Wahl praktisch folgenlos ist, liegt daran, dass eine einzelne Stelle beim Lernen aus endlich vielen Datenpunkten fast nie exakt getroffen wird. Die Konsequenzen für die Wahl der Aktivierungsfunktion stehen im [Kapitel „Aktivierungsfunktionen“](#).

Weiterlesen

Deisenroth et al., *MML*, Kap. 5 (Vector calculus: partielle Ableitungen, Gradienten, Jacobi- und Hesse-Matrix). Frei verfügbar, siehe [Über dieses Skript](#).

2.4 Neuron und Perzeptron

2.4.1 Das Neuron und das Perzeptron

Das McCulloch-Pitts-Neuron

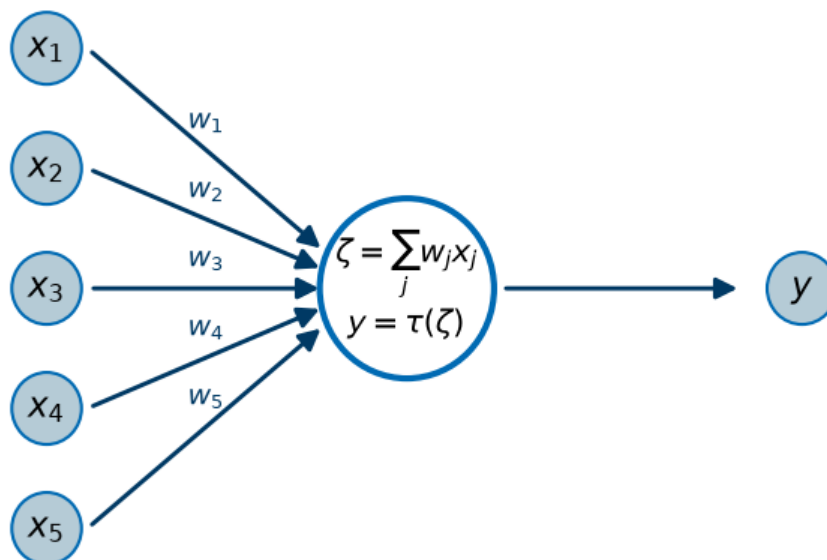
Das Modell von 1943 ist von der Biologie inspiriert: Ein Neuron bekommt über Dendriten Signale $x_1, \dots, x_n$ von anderen Neuronen, summiert sie gewichtet auf und „feuert“, wenn die Summe eine Schwelle überschreitet. Als Formel: Mit Gewichten $w_1, \dots, w_n \in \mathbb{R}$ ist die *Aktivierung*

$$\zeta = \sum_{j=1}^n w_j x_j = w \cdot x \quad (15)$$

und die Ausgabe

$$y = \tau(\zeta) \quad (16)$$

für eine nichtlineare *Aktivierungsfunktion* $\tau: \mathbb{R} \rightarrow \mathbb{R}$. Bei McCulloch und Pitts ist τ eine Schwellenwertfunktion, die 0 oder 1 ausgibt; später wird τ die logistische Sigmoidfunktion σ sein oder ReLU. Diese Rollenverteilung gilt im ganzen Buch: τ ist eine beliebige Aktivierungsfunktion, σ immer die logistische Sigmoidfunktion. Die Struktur „Skalarprodukt, gefolgt von einer nichtlinearen Funktion“ ist das Kochrezept, das uns durch die gesamte Vorlesung begleiten wird.



Das McCulloch-Pitts-Neuron: gewichtete Summe der Eingaben, dann eine nichtlineare Aktivierungsfunktion.

Das Perzeptron

Rosenblatts Beitrag war weniger das Neuron als der *Lernalgorithmus* dafür. Das Perzeptron ist ein Modell zur **binären Klassifikation**: Input sind Stichproben von (X, Y) mit $X \in \mathbb{R}^n$ und $Y \in \{0, 1\}$. Die Parameter des Modells sind ein Gewichtsvektor $w \in \mathbb{R}^n$ und ein Biasterm $b \in \mathbb{R}$. Die Inferenz ist

$$f(x) = [w \cdot x + b > 0], \quad (17)$$

wobei $[\cdot]$ die Iverson-Klammer ist: $[A] = 1$, wenn die Aussage A wahr ist, sonst 0. Das Perzeptron ist also ein McCulloch-Pitts-Neuron mit der Schwellenwertfunktion $\tau(\zeta) = [\zeta + b > 0]$. Man beachte: f ist eine *nichtlineare* Funktion, obwohl im Inneren nur ein Skalarprodukt steht.

Den Bias kann man loswerden, indem man ihn als nulltes Gewicht auffasst: Mit $w_0 := b$ und $x_0 := 1$ ist $w \cdot x + b = \sum_{j=0}^n w_j x_j$. Das ist ein Trick, den wir noch öfter sehen werden. Umgekehrt heißt das: Ein

Perzeptron *ohne* Bias kann nur Hyperebenen durch den Ursprung lernen – zwei Klassen, die sich nur mit einer verschobenen Geraden trennen lassen, bleiben ihm verschlossen.

Geometrisch beschreibt die Gleichung $w \cdot x + b = 0$ eine Hyperebene H in $\mathbb{R}^n$ (in $\mathbb{R}^2$: eine Gerade), und f gibt an, auf welcher Seite von H der Punkt x liegt. Der Vektor w steht senkrecht auf H : Für zwei Punkte $x, x' \in H$ ist

$$w \cdot (x - x') = (w \cdot x + b) - (w \cdot x' + b) = 0 - 0 = 0, \quad (18)$$

also ist w orthogonal zu jeder Richtung *innerhalb* von H .

Mehr noch: Die Aktivierung $w \cdot x + b$ misst, *wie weit* x von H entfernt ist. Sei $\tilde{w} = w/\|w\|$ der normierte Normalenvektor. Jeder Punkt lässt sich zerlegen als $x = x_H + d \tilde{w}$ mit einem Punkt $x_H \in H$ (dem Lotfußpunkt) und einer Zahl $d \in \mathbb{R}$. Dann ist

$$w \cdot x + b = (w \cdot x_H + b) + d(w \cdot \tilde{w}) = 0 + d\|w\|, \quad \text{also} \quad d = \frac{w \cdot x + b}{\|w\|}. \quad (19)$$

Das ist der **vorzeichenbehaftete Abstand** von x zur Hyperebene: positiv auf der Seite, in die w zeigt, negativ auf der anderen, null auf H . Die Aktivierung eines Neurons ist nichts anderes als dieser Abstand, multipliziert mit $\|w\|$.

Example 2.10 (Abstand zur Trenngeraden). Sei $w = (3, 4)$ und $b = -5$, also $\|w\| = 5$ und $H = \{x \in \mathbb{R}^2 : 3x_1 + 4x_2 = 5\}$. Der Punkt $(1, \frac{1}{2})$ liegt auf H , denn $3 + 2 - 5 = 0$. Der Punkt $(3, 4)$ hat Aktivierung $9 + 16 - 5 = 20$ und damit Abstand $20/5 = 4$. Der Punkt $(-1, 0)$ hat Aktivierung $-3 - 5 = -8$ und Abstand $-8/5$: andere Seite.

Daraus folgt eine Beobachtung, die uns bei der logistischen Regression wieder begegnen wird. Multipliziert man w und b mit derselben Zahl $c > 0$, so ändert sich weder H noch die Entscheidung $[w \cdot x + b > 0]$, aber die Aktivierung wird um den Faktor c skaliert. Die *Richtung* von w legt fest, *was* klassifiziert wird; die *Länge* von w legt fest, wie schnell die Aktivierung mit dem Abstand wächst. Für das Perzeptron ist die Länge egal. Sobald wir statt der Sprungfunktion eine glatte Aktivierungsfunktion verwenden, ist sie es nicht mehr: Für die logistische Sigmoidfunktion σ ist $\sigma(c \cdot 20)$ bei $c = 10$ praktisch 1, bei $c = 0,01$ nur knapp über $\frac{1}{2}$. Dieselbe Trenngerade, ganz andere „Sicherheit“.

Definition 2.18 (Linear trennbar). Zwei Mengen $A, B \subseteq \mathbb{R}^n$ heißen *linear trennbar*, wenn es $w \in \mathbb{R}^n$ und $b \in \mathbb{R}$ gibt mit $w \cdot a + b > 0$ für alle $a \in A$ und $w \cdot a' + b \leq 0$ für alle $a' \in B$.

Ein Perzeptron kann also genau die Klassifikationsprobleme lösen, bei denen die beiden Klassen linear trennbar sind – nicht mehr. Das berühmte Gegenbeispiel ist XOR: Die Punkte $(0, 0)$ und $(1, 1)$ lassen sich nicht durch eine Gerade von $(0, 1)$ und $(1, 0)$ trennen. Das ist die mathematische Substanz hinter dem AI Winter – und der Grund, warum wir mehrere Neuronen hintereinander schalten werden.

Wie das Perzeptron *trainiert* wird, also wie w und b aus den Daten bestimmt werden, ist Thema des [nächsten Kapitels](#); auf einem Übungsblatt implementieren Sie es dann selbst.

2.4.2 Was ist ein neuronales Netz?

Ein **Multilayer Perceptron** (MLP) entsteht, indem die Ausgaben von mehreren (verallgemeinerten) Perzeptrons als Eingabe für ein weiteres Perzeptron verwendet werden, welches also eine Schicht (Layer) weiter hinten ist.

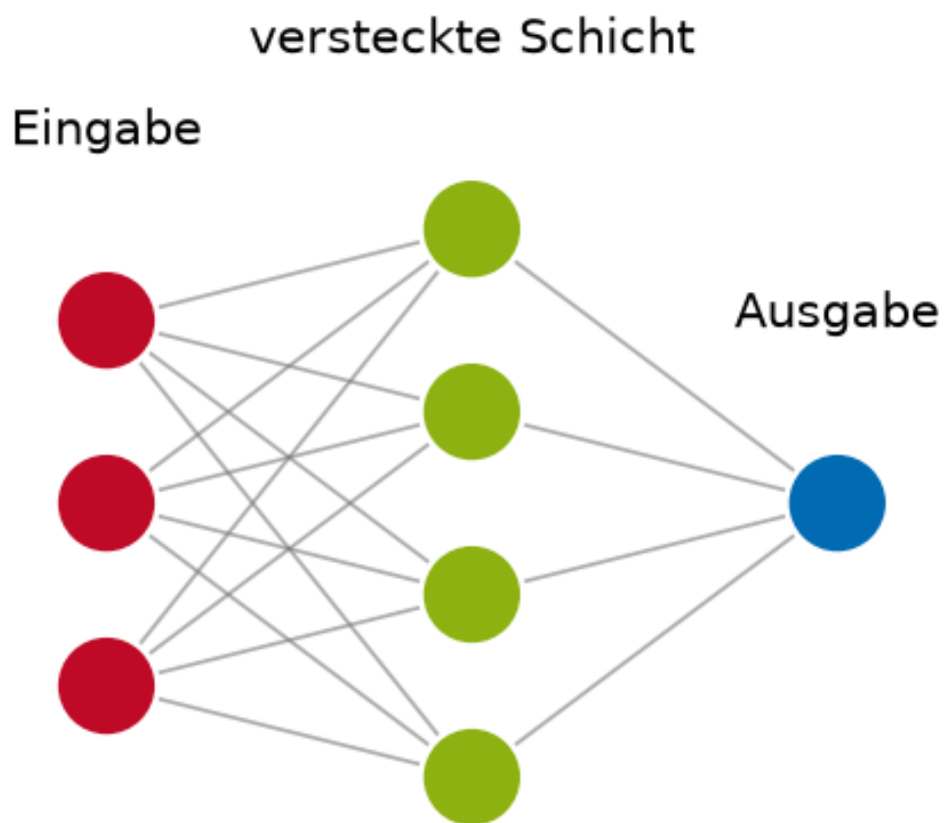
Formal: Eine Schicht mit m Neuronen, die alle dieselben n Eingaben bekommen, hat eine Gewichtsmatrix $W \in \mathbb{R}^{m \times n}$ und einen Biasvektor $b \in \mathbb{R}^m$ und berechnet

$$h = \tau(Wx + b), \quad (20)$$

wobei τ komponentenweise angewendet wird.

Wie liest man Wx ? Die i -te Komponente von Wx ist das Skalarprodukt der i -ten Zeile von W mit x . Die Zeilen von W (in NumPy `W[i, :]`) sind also die Gewichtsvektoren der m Neuronen, und $Wx + b$ berechnet m Aktivierungen auf einmal, mit dem vorigen Abschnitt: m skalierte Abstände zu m Hyperebenen. Als Bild in $\mathbb{R}^2$: Jedes Neuron der ersten Schicht zeichnet eine Gerade in die Eingabeebene, und der Vektor h sagt für jede dieser Geraden, auf welcher Seite x liegt und wie weit weg. Das XOR-Beispiel unten ist genau das: zwei Geraden, dann eine Entscheidung im Raum der beiden Seiten-Informationen.

In PyTorch speichert `nn.Linear(n, m)` die Matrix als `weight` mit Shape `(m, n)`, eine Zeile pro Neuron, und `bias` mit Shape `(m,)`.



Ein Multilayer Perceptron mit 3 Eingaben, einer versteckten Schicht aus 4 Neuronen und einem Ausgabeneuron. Jeder Pfeil ist ein Gewicht.

Ein Netz mit L Schichten berechnet dann

$$h^{(0)} = x, \quad h^{(k)} = \tau^{(k)}(W^{(k)}h^{(k-1)} + b^{(k)}) \quad (k = 1, \dots, L), \quad f(x) = h^{(L)}. \quad (21)$$

Dabei können die Schichten verschieden groß sein, manche Einträge von $W^{(k)}$ können 0 sein (fehlende Pfeile), es kann Pfeile rückwärts geben (rekurrente Netze), und es können viele verschiedene nichtlineare Funktionen vorkommen (ReLU, Convolution, Softmax, ...). Die Gesamtheit der Einträge aller $W^{(k)}$ und $b^{(k)}$ sind die *Parameter* oder *Gewichte* des Netzes; bei modernen Modellen sind das Milliarden von Zahlen. „Deep“ heißt: L ist groß.

Warum das Hintereinanderschalten hilft, sehen wir am XOR-Problem: Mit zwei Perzeptronen in der ersten Schicht, die „ $x_1 \vee x_2$ “ und „ $\neg(x_1 \wedge x_2)$ “ berechnen, und einem dritten Perzeptron, das deren Ausgaben mit UND verknüpft, ist XOR plötzlich linear trennbar – im Raum der Ausgaben der ersten Schicht.

2.4.3 Übung

Sei $w = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ und $b = -5$, also das Perzeptron $x \mapsto [w \cdot x + b > 0]$.

1. Berechnen Sie für $x_1 = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$, $x_2 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$ und $x_3 = \begin{pmatrix} 3 \\ -1 \end{pmatrix}$ jeweils die Aktivierung $w \cdot x + b$ **und** den vorzeichenbehafteten Abstand zur Geraden. *Tipp: Der Abstand ist die Aktivierung geteilt durch $\|w\|$, und $\|w\| = 5$.*
2. Ersetzen Sie w durch $10w$ und b durch $10b$. Was ändert sich an der Geraden, was an den Aktivierungen, was an den Abständen?

Antwortprobe: Genau eine der drei Größen ändert sich.

Weiterlesen

Prince, *UDL*, Kap. 3 (Shallow neural networks); Deisenroth et al., *MML*, Kap. 3 (Analytic geometry: Skalarprodukt, Winkel, Projektion). Beide Bücher sind frei verfügbar, siehe [Über dieses Skript](#).

2.5 Das Perzeptron trainieren

2.5.1 Das Perzeptron trainieren

Im [letzten Kapitel](#) haben wir das Perzeptron als Modell kennengelernt: ein Gewichtsvektor $w \in \mathbb{R}^n$, ein Bias $b \in \mathbb{R}$, und die Inferenz $f(x) = [w \cdot x + b > 0]$. Was fehlt, ist das *Lernen*. Bevor wir ein Modell implementieren, sollten wir uns ein paar Fragen stellen, die für jedes Modell in dieser Vorlesung wieder auftauchen werden:

- Was sind die **freien Parameter** (Gewichte) des Modells – welche Variablen ändern sich beim Training?
- Wie soll die **Inferenz** durchgeführt werden?
- Wie funktioniert das **Training**?
 - supervised oder unsupervised?
 - Was genau erlauben wir für Datentypen?
 - Welche Berechnung passt die Gewichte an?
 - Was ist die Zielgröße – wann ist genug trainiert?
 - Welche **Hyperparameter** gibt es, und worauf legen wir sie fest?

Für das Perzeptron beantworten wir sie so.

Daten und Aufgabe. Das Perzeptron ist ein Modell zur binären Klassifikation, also Supervised Learning. Input sind Stichproben von (X, Y) , wobei X Werte in $\mathbb{R}^n$ hat und $Y \in \{0, 1\}$ die Klasse ist. Die Dimensionalität n der Eingabedaten ist ein Hyperparameter – einer, den wir in der Regel nicht optimieren wollen, weil er von den Daten vorgegeben ist. Ziel ist die Berechnung einer Funktion f mit $Y = f(X) + \varepsilon$ für einen möglichst kleinen Fehler ε .

Parameter und Inferenz. Die Parameter sind n Gewichte w_j , zusammengefasst zu $w \in \mathbb{R}^n$, und der Bias b . Die Inferenz ist

$$f(x) = [w \cdot x + b > 0], \quad (22)$$

also $f(x) = 1$ genau dann, wenn $\sum_j w_j x_j + b > 0$. Mit dem Trick $w_0 := b$, $x_0 := 1$ aus dem letzten Kapitel schreiben wir im Folgenden einfach $f(x) = [w \cdot x > 0]$ und denken uns den Bias als nulltes Gewicht.

Training: die Perzeptronregel. Gegeben die bisherigen Gewichte w und ein einzelnes Sample (x, y) , setze

$$w_{\text{neu}} := w + \rho \cdot (y - f(x)) x. \quad (23)$$

Dabei ist $\rho > 0$ ein Hyperparameter, die **Lernrate**. Was macht diese Regel?

- Wenn $y = f(x)$, also bei korrekter Klassifikation, ist $y - f(x) = 0$ und damit $w_{\text{neu}} = w$: Nichts passiert.
- Wenn $y = 1$, aber $f(x) = 0$, so ist $w_{\text{neu}} = w + \rho x$. Dann ist $w_{\text{neu}} \cdot x = w \cdot x + \rho(x \cdot x)$, und weil $x \cdot x > 0$ für alle $x \neq 0$ gilt, ist die Aktivierung für dieses x *gewachsen* – in Richtung der richtigen Entscheidung $f(x) = 1$.
- Wenn $y = 0$, aber $f(x) = 1$, so ist $w_{\text{neu}} = w - \rho x$, und die Aktivierung *sinkt* entsprechend.

Warum funktioniert das nicht nur für einen Schritt, sondern irgendwann sicher? Für ein festes Sample (x, y) mit $y = 1$, das nach k Updates immer noch falsch klassifiziert ist, gilt wegen der Bilinearität des Skalarprodukts

$$w^{(k)} \cdot x = w^{(0)} \cdot x + k \rho \|x\|^2, \quad (24)$$

und dieser Wert wird für großes k irgendwann positiv, egal wie schlecht $w^{(0)}$ war. Geometrisch: Jedes Update addiert ein Vielfaches von x zu w , dreht w also Schritt für Schritt *in Richtung* x (bei $y = 0$ von x weg), und die Entscheidungsgrenze, die senkrecht auf w steht, dreht sich mit, bis x auf der richtigen Seite liegt. Das ist die eine Rechnung, die man zu diesem Kapitel einmal selbst auf Papier gemacht haben

sollte. Natürlich kann ein Update für x ein anderes Sample x' auf die falsche Seite schieben; dass sich das nicht endlos wiederholt, ist die Aussage des Konvergenzsatzes.

Die Regel schiebt also die Trennhyperebene bei jedem Fehler ein Stück in Richtung des falsch klassifizierten Punkts. Falls die Daten linear trennbar sind, findet sie nach endlich vielen Fehlern eine trennende Hyperebene; falls nicht, terminiert das Verfahren nie von selbst.

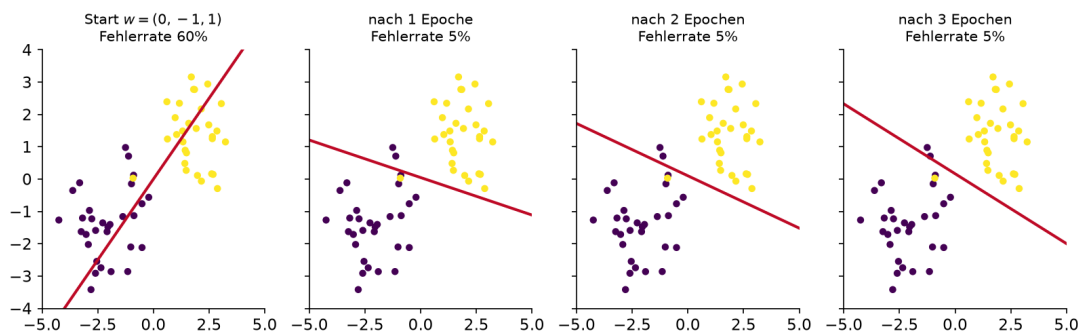
Vertiefung: der Perzeptron-Konvergenzsatz

(Novikoff 1962.) Seien die Label als $y_i \in \{-1, +1\}$ kodiert. Angenommen, es gibt einen Einheitsvektor w^* und eine *Margin* $\gamma > 0$ mit $y_i (w^* \cdot x_i) \geq \gamma$ für alle i , und alle Datenpunkte erfüllen $\|x_i\| \leq R$. Dann macht die Perzeptronregel mit Start $w^{(0)} = 0$ höchstens R^2/γ^2 Fehler, unabhängig von der Lernrate, der Dimension n und der Anzahl N der Datenpunkte.

Beweisidee: Bei jedem Fehler wächst $w \cdot w^*$ um mindestens $\rho\gamma$, nach k Fehlern also auf mindestens $k\rho\gamma$. Gleichzeitig wächst $\|w\|^2$ bei jedem Fehler um höchstens $\rho^2 R^2$, denn $\|w + \rho y x\|^2 = \|w\|^2 + 2\rho y (w \cdot x) + \rho^2 \|x\|^2$, und der mittlere Term ist bei einem Fehler nicht positiv. Nach k Fehlern ist also $\|w\| \leq \rho R \sqrt{k}$. Mit Cauchy-Schwarz $w \cdot w^* \leq \|w\|$ folgt $k\rho\gamma \leq \rho R \sqrt{k}$, also $k \leq R^2/\gamma^2$. Das Verfahren nutzt die Margin, ohne sie zu kennen.

Zielgröße. Wir messen den Erfolg als Anteil falsch klassifizierter Samples (Fehlerrate), bzw. als Anteil korrekt klassifizierter Samples (Accuracy) auf einem Testset.

Initialisierung. Irgendwo muss w anfangen; wir initialisieren w z.B. auf den Vektor $(1, 1, \dots, 1)$. Auch das ist eine Entscheidung mit Konsequenzen, wie wir später sehen werden.



Die Perzeptronregel bei der Arbeit: Trenngerade $w \cdot x + b = 0$ nach 0, 1, 2 und 3 Epochen über 60 Datenpunkte, Lernrate $\rho = 0,05$.

2.5.2 Zurück in die Theorie: Wozu das olle Perzeptron?

Moderne neuronale Netze verwenden meist keine Perzeptrons mehr. Beliebter ist die multiple lineare oder die logistische Regression, die ebenfalls gleich strukturierte Eingabedaten annimmt und sehr ähnlich verarbeitet: *Skalarprodukt, gefolgt von einer nichtlinearen Funktion*. Dieses Kochrezept – die gemeinsame Verallgemeinerung – heißt manchmal auch einfach „Perzeptron“.

Definition 2.19 (Verallgemeinertes Perzeptron). *Das verallgemeinerte Perzeptron ist ein Regressionsmodell mit Parametern $w \in \mathbb{R}^n$ und Funktion*

$$f(x) = \tau(w \cdot x) \quad (25)$$

für eine feste Aktivierungsfunktion $\tau: \mathbb{R} \rightarrow \mathbb{R}$.

Die Wahl von τ bestimmt, welches bekannte Modell wir vor uns haben – und wie man es trainiert:

Modell	$\tau(a)$	Training
Lineare Regression	$\tau(a) = a$	kleinste Quadrate
Perzeptron	$\tau(a) = [a > 0]$	Perzeptronregel $w_{\text{neu}} := w + \rho(y - f(x))x$
„stetiges Perzeptron“	$\tau(a) = \tanh(a)$	?
Logistische Regression	$\tau(a) = \sigma(a) = \frac{1}{1+e^{-a}}$	?

Die letzten beiden Zeilen sind stetige, sogar differenzierbare Versionen der Sprungfunktion $[a > 0]$: $\tanh$ bildet nach $(-1, 1)$ ab, die logistische Sigmoidfunktion σ nach $(0, 1)$. Die offene Frage ist die Update-Regel: Wie trainiert man ein verallgemeinertes Perzeptron mit beliebigem τ ? Die Antwort – Gradientenabstieg auf einer Loss-Funktion – werden wir in den nächsten Kapiteln eingehend studieren; die Perzeptronregel wird sich dabei als Spezialfall erweisen.

2.5.3 Mehrere Schichten

Warum die Perzeptronregel nicht reicht

Angenommen, $x \in \mathbb{R}^2$ und $v, w \in \mathbb{R}^2$ sind die Gewichte von zwei Perzeptrons,

$$g(x) = [v \cdot x > 0], \quad f(x) = [w \cdot x > 0], \quad (26)$$

und $z := (g(x), f(x)) \in \{0, 1\}^2$ ist der Ergebnisvektor der ersten Schicht. Ein drittes Perzeptron mit Gewichten $u \in \mathbb{R}^2$ berechnet daraus

$$h(x) = [u \cdot z > 0]. \quad (27)$$

Das ist ein Multilayer Perceptron mit einer versteckten Schicht. Die Perzeptronregel sagt uns, wie wir u anpassen, wenn $h(x)$ falsch liegt – aber sie sagt nichts darüber, wie v und w angepasst werden sollten. Welches der beiden Perzeptrons in der ersten Schicht war „schuld“? Die Antwort auf diese Frage heißt **Backpropagation**, und sie braucht ein τ , das sich ableiten lässt. Die Sprungfunktion des Perzeptrons tut das nicht; ReLU dagegen wird überall benutzt, obwohl sie bei 0 einen Knick hat (siehe [Kapitel „Aktivierungsfunktionen“](#)). Das machen wir noch.

Warum lineare Regression keine Schichten braucht

Man könnte auf die Idee kommen, statt Perzeptrons einfach lineare Regressionen zu stapeln, die haben ja ein bekanntes Trainingsverfahren. Seien also $g(x) = v \cdot x$ und $f(x) = w \cdot x$ zwei lineare Regressionen, $z := (v \cdot x, w \cdot x)$, und $h(x) = u \cdot z$. Dann ist

$$h(x) = u_1(v \cdot x) + u_2(w \cdot x) = u_1v_1x_1 + u_1v_2x_2 + u_2w_1x_1 + u_2w_2x_2 = (u_1v + u_2w) \cdot x. \quad (28)$$

Das ist äquivalent zu einer einzigen linearen Regression mit Gewichtsvektor $u_1v + u_2w$. Allgemein: Ein Produkt von Matrizen ist wieder eine Matrix, mehrere lineare Schichten hintereinander sind wieder eine lineare Schicht. Ohne die Nichtlinearität τ zwischen den Schichten bringt Tiefe gar nichts. Genau deshalb heißt τ auch *Aktivierungsfunktion* und nicht Dekoration.

2.5.4 Übung

Gegeben ein Datenpunkt $x = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$ mit Label $y = 1$ und ein Perzeptron mit $w^{(0)} = \begin{pmatrix} -3 \\ 1 \end{pmatrix}$, also $w^{(0)} \cdot x = -5 < 0$: Das Perzeptron sagt 0, richtig wäre 1.

1. Führen Sie einen Schritt der Perzeptronregel $w^{(1)} = w^{(0)} + \rho(y - f(x))x$ mit $\rho = 0,5$ aus.
2. Berechnen Sie $w^{(1)} \cdot x$. Um wie viel hat sich die Aktivierung geändert? *Tipp: Rechnen Sie die Differenz einmal allgemein mit der Bilinearität des Skalarprodukts aus – Sie bekommen $\rho \|x\|^2$, unabhängig von $w^{(0)}$.*
3. Wie viele solche Schritte braucht es mindestens, bis die Aktivierung positiv wird?

Antwortprobe: eine ganze Zahl, ohne Iteration ausrechenbar. Machen Sie die Gegenprobe mit einem Schritt weniger – dort muss die Aktivierung noch ≤ 0 sein, denn in $[w \cdot x > 0]$ steht ein echtes Größ-erzeichen.

Weiterlesen

Prince, *UDL*, Kap. 3 (Shallow neural networks) und 4 (Deep neural networks) für das Stapeln von Schichten. Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).

2.6 Vortrainierte Modelle verwenden

2.6.1 Orientierung

Was tun mit Deep Learning?

Bevor wir selbst Modelle trainieren, lohnt sich ein Blick darauf, was es schon gibt. Wenn wir für eine Aufgabe ein vortrainiertes Modell verwenden wollen, müssen wir wissen, welches Modell geeignet sein kann. Dafür schauen wir bei **Hugging Face**: huggingface.co/models listet Modelle, huggingface.co/tasks sortiert sie nach Aufgabe („The Hugging Face Hub hosts many models for a variety of machine learning tasks.“). Die Task-Seiten sind ein guter Einstieg, um das Vokabular zu lernen: Was ist Image Classification, Object Detection, Token Classification, Text Generation, ...? Einen systematischen Rundgang machen wir im nächsten Abschnitt.

Die Task-Seite gruppiert nach Modalität. Zur Orientierung, in der Reihenfolge von dort:

Modalität	Tasks (Auswahl)
Computer Vision	Image Classification, Object Detection, Image Segmentation, Depth Estimation, Image-to-Image, Text-to-Image, Zero-Shot Image Classification
Natural Language Processing	Text Classification, Token Classification, Question Answering, Summarization, Translation, Text Generation, Fill-Mask, Sentence Similarity
Audio	Automatic Speech Recognition, Text-to-Speech, Audio Classification
Multimodal	Image-to-Text, Visual Question Answering, Document Question Answering, Feature Extraction
Tabular, Reinforcement Learning	Tabular Classification und Regression; Reinforcement Learning

Die Namen sind das Vokabular, in dem Modellkarten, Papers und Stellenanzeigen reden.

Hugging Face ...?

Surfen Sie ruhig weiter auf huggingface.co. Hier ein paar Fakten zur Einordnung:

- HF verdient Geld mit gehosteter Inferenz.
- HF lockt Entwickler an mit guter Infrastruktur: Libraries (z.B. **transformers**, **diffusers**, **gradio**), Hosting (git-Repositories für Modelle, Datensätze und „Spaces“) und Dokumentation – insbesondere für **Finetuning**, das Nachtrainieren von vortrainierten Modellen.
- HF ist ein Unternehmen aus New York City, gegründet 2016. In der Finanzierungsrunde vom August 2023 war es 4,5 Mrd. US-Dollar wert; gut drei Jahre später, im September 2026, hat **Nvidia** den Kauf für 12,93 Mrd. US-Dollar vereinbart – der Abschluss wird für die erste Hälfte 2027 erwartet, vorbehaltlich der Freigaben. Die Plattform, auf der die offenen Gewichte liegen, gehört damit demselben Konzern wie die Hardware, auf der sie gerechnet werden.

Mit anderen Worten: Der Hub ist ein kommerzielles Angebot, aber die Modelle und Libraries darauf sind größtenteils offen, und für uns ist er die bequemste Art, an Gewichte zu kommen.

2.6.2 Ein Rundgang durch die Tasks

Wenn wir für eine Aufgabe ein vortrainiertes Modell verwenden wollen, müssen wir wissen, welches Modell geeignet sein kann. Auf huggingface.co/tasks sind die Modelle nach *Tasks* sortiert. Wir gehen exemplarisch einige davon durch – nicht, um die Modelle zu verstehen, sondern um das Vokabular zu lernen und ein Gefühl dafür zu bekommen, was heute „von der Stange“ geht.

Tasks in Computer Vision

- **Depth Estimation**: Von einem Bild für jeden Pixel bei (x, y) auch eine z -Koordinate schätzen. Beispiel: Intels *Dense Prediction Transformer* (DPT) von 2021, ein Vision Transformer.
- **Image Classification**: Von einem Bild sagen, was es ist. Beispiel: Microsofts *ResNet* (CNN, 2016), trainiert auf ImageNet – gibt zu einem Foto „Igel“ zurück.

- **Object Detection:** Von einem Bild sagen, was drauf ist, mit Bounding Boxes. Beispiel: *YOLOS* (NeurIPS 2021), trainiert auf ImageNet und COCO – findet in Bild #7781 des COCO-Datensatzes „Cat, Banana, Chair“.
- **Image Segmentation:** Auf einem Bild für jedes erkannte Objekt die Umrisse finden, das Bild also aufteilen. Beispiel: Nvidias *Segformer* (2021), auf ImageNet vortrainiert und auf ADE20k gefinetunt.
- **Image-to-Image:** Ein Bild als Input, ein Bild als Output, z.B. historische Fotos einfärben. Beispiel: Renato Sortinos *ColorizeNet*, ein Finetune von ControlNet / Stable Diffusion 2.1 auf bearbeitetem COCO.
- **Unconditional Image Generation:** Nur ein Bild als Output, z.B. ein Katzenbild (je nach Datenbasis). Beispiel: *DDPM* (Ho et al., 2020), Denoising Diffusion Probabilistic Models – auf dem Hub liegt der Checkpoint unter [google/](https://huggingface.co/CompVis/stable-diffusion-v1-4), das Verfahren kommt aus Berkeley.
- **Zero-Shot Image Classification:** Von einem Bild sagen, was es ist – ohne dass für die Klassen gelabelte Daten vorlagen. Beispiel: OpenAIs *CLIP-ViT* (2021), Contrastive Language-Image Pretrained Vision Transformer, dem man die Kandidaten als Text gibt („a photo of a car“).

Zur Einordnung des Vokabulars: **Zero-Shot** heißt ohne gelabelte Daten für die neue Aufgabe, **One-Shot** (oder Single-Shot) mit einem gelabelten Sample, **Few-Shot** mit einigen wenigen gelabelten Samples.

Tasks in NLP

Wir betrachten nur einen Teil der NLP-Tasks:

- **Text Classification:** Texten ein Genre, eine Stimmung (Sentiment) o.ä. zuordnen. Beispiel: *DistilBERT* (2019), self-supervised trainiert mit BERT (Bidirectional Encoder Representations from Transformers) als Lehrer – zur [Destillation](#) kommen wir gegen Ende der Vorlesung.
- **Token Classification:** Named Entity Recognition (NER) oder Part-of-Speech (PoS) Tagging, also jedem Wort ein Label geben. Beispiel: *flair ner-german* (2020) von der HU Berlin.
- **Summarization:** Zu einem Text (Satz oder Dokument) eine kurze Zusammenfassung schreiben. Beispiel: Facebooks *BART* (2019), Bidirectional and Auto-Regressive Transformers.
- **Fill-Mask:** Einen Lückentext füllen. Beispiel: wieder *DistilBERT* – das ist nämlich genau die Aufgabe, mit der BERT-artige Modelle vortrainiert werden.

2.6.3 Praktisch

Beispiel: Übersetzen mit WMT19

Wir übersetzen von Englisch nach Deutsch mit Facebooks WMT19-Modell, nach der Anleitung von Hugging Face:

```
from transformers import FSMTForConditionalGeneration, FSMTTokenizer

mname = "facebook/wmt19-en-de"
tokenizer = FSMTTokenizer.from_pretrained(mname)
model = FSMTForConditionalGeneration.from_pretrained(mname)

input = "I am not sure if deep learning is really for me."
input_ids = tokenizer.encode(input, return_tensors="pt")
outputs = model.generate(input_ids)
decoded = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(decoded)
# Ich bin mir nicht sicher, ob Deep Learning wirklich etwas für mich ist.
```

Das Muster ist typisch für alle Modelle auf dem Hub: Ein **Tokenizer** verwandelt Text in Zahlen (`input_ids`, als PyTorch-Tensor, daher `return_tensors="pt"`), das **Modell** verwandelt Zahlen in Zahlen, und der Tokenizer verwandelt das Ergebnis zurück in Text. `from_pretrained` lädt die Gewichte beim ersten Aufruf aus dem Hub herunter und cached sie lokal.

Auf der GPU rechnen

Um auf der GPU mit CUDA zu rechnen, müssen Modell *und* Eingabedaten dorthin verschoben werden:

```
model = FSMTForConditionalGeneration.from_pretrained(mname).to("cuda")
input_ids = tokenizer.encode(input, return_tensors="pt").to("cuda")
```

Der übliche Workflow, damit der Code auch ohne GPU läuft:

```
import torch

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)
```

Alternativ kann die **accelerate**-Library von HF auch dynamisch Teile des Modells auf die GPU legen und andere Teile im Hauptspeicher belassen, um sehr große Modelle zu nutzen.

Was zuvor geschah: Installation

`pip install 'transformers[torch]'` in der Shell (oder `!pip install ...` in einem Notebook) installiert **transformers** und PyTorch. Das allein installiert noch kein CUDA, d.h. alle Rechnungen finden auf der CPU mit dem Hauptspeicher statt – in PyTorch ist das schlicht das Gerät **cpu**, ohne Indexnummer. In den **pipeline**-Aufrufen von **transformers** schreibt man dafür `device=-1`; `device=0` meint dort die *erste GPU*.

Für CUDA (Ubuntu Linux auf NVIDIA-Hardware) prüft man zuerst, was da ist:

```
lspci | grep -i nvidia
nvidia-smi
```

nvidia-smi zeigt die Treiberversion und die maximale CUDA-Version, die die GPU bzw. der Treiber unterstützt (z.B. 11.4); **nvcc --version** zeigt die installierte CUDA-Toolkit-Version (z.B. 10.1). Die installierte Version darf kleiner sein als die maximale – das passt dann. Anschließend PyTorch mit passendem CUDA-Build installieren:

```
pip install torch torchvision --index-url https://download.pytorch.org/whl/cu129
```

Hat es geklappt? Python-Interpreter auf und `torch.cuda.is_available()` sollte **True** liefern. Auf AMD-GPUs heißt das Ganze ROCm statt CUDA, aber der Python-Befehl ist derselbe; auf macOS 12.3+ gibt es Beschleunigung über MPS (`torch.backends.mps.is_available()`).

Und noch davor: eine virtuelle Umgebung im Arbeitsordner anlegen, damit die Installation nicht das System-Python verändert:

```
python -m venv .venv
source .venv/bin/activate
```

und erst *danach* mit **pip** fleißig installieren. Oder entspannt **uv** benutzen, das Umgebung und Pakete in einem Schritt verwaltet.

Alternative: Google Colab

Wer nicht lokal installieren will oder keine GPU hat: **Google Colab** bietet Jupyter-Notebooks mit (begrenzter) GPU-Zeit im Browser. Ein neues Notebook gibt es unter colab.research.google.com/#create=true; die Hugging-Face-Dokumentation ist als Colab-Notebooks verfügbar, z.B. die [Task Summary](#).

2.7 Tensoren

2.7.1 Tensoren in PyTorch

Tensoren $\approx$ Arrays

Vier Fragen: Was ist ein Array? Was ist ein Tensor? Was ist ein Array in NumPy? Was ist ein Tensor in Torch?

- Ein **Array** ist eine Sammlung von Daten von gleichem Typ, die fortlaufend im Speicher liegen (können).
- Ein **Tensor** ist, je nachdem, wen man fragt:
 - Physiker: ein Objekt, das sich tensoriell transformiert.
 - Mathematiker: ein Element in einem Tensorproduktraum.
 - Informatiker: ein fancy Wort für Arrays.
- Ein **Tensor in Torch** ist eine Sammlung von Daten von gleichem Typ (dem `dtype`), die fortlaufend im Speicher liegen – und zwar potentiell im GPU-Speicher – optimiert für Gradientenabstiegsmethoden (Attribut `grad`, dazu mehr in den Kapiteln „[Gradientenabstieg](#)“ und „[Autograd und Berechnungsgraph](#)“).

Man kann einen Torch-Tensor aus einem NumPy-Array herstellen, sogar ohne zusätzlichen Speicher zu verbrauchen (`torch.from_numpy` teilt sich den Speicher mit dem Array). Lesen Sie dazu das PyTorch-Tutorial „Tensors“.

Ausflug: Tensoren in der Mathematik

Der Informatiker hat nicht ganz Unrecht, aber das mathematische Konzept hilft zu verstehen, was `shape` und `reshape` eigentlich bedeuten. Wer den Ausflug überspringen will, kann bei „Ausflug beendet“ weiterlesen.

Tensorprodukt. Gegeben Vektorräume U, V, W (z.B. $V = \mathbb{R}^n$ und $W = \mathbb{R}^m$) kann man das Tensorprodukt bilden: $V \otimes W$ ist definiert als ein Vektorraum, sodass es eine 1:1-Korrespondenz gibt

$$\{\text{bilineare Abbildungen } V \times W \rightarrow U\} \longleftrightarrow \{\text{lineare Abbildungen } V \otimes W \rightarrow U\}. \quad (29)$$

Insbesondere gibt es eine bilineare Abbildung $V \times W \rightarrow V \otimes W$, $(v, w) \mapsto v \otimes w$. Wenn $v_1, \dots, v_n \in V$ und $w_1, \dots, w_m \in W$ Basen sind, so bilden die $v_i \otimes w_j$ eine Basis von $V \otimes W$; insbesondere ist $\dim(V \otimes W) = n \cdot m$ und $\mathbb{R}^n \otimes \mathbb{R}^m \cong \mathbb{R}^{nm}$.

Dualraum. Man kann den Dualraum bilden: $V^\vee := \text{Hom}(V, \mathbb{R})$, die Menge der linearen Abbildungen $V \rightarrow \mathbb{R}$, ist ein neuer Vektorraum. Man nennt die Elemente von V *Vektoren*, die Elemente von $V^\vee$ *Kovektoren* oder *Linearformen*. Wenn $v_1, \dots, v_n$ eine Basis von V ist, bilden die Linearformen $v_1^\vee, \dots, v_n^\vee$ mit $v_i^\vee(v_j) = 1$ für $i = j$ und $v_i^\vee(v_j) = 0$ sonst eine Basis von $V^\vee$, die *duale Basis*. Ein Skalarprodukt braucht man dafür nicht. Man sagt, die Elemente von $W \otimes V^\vee$ sind *kontravariant* in W und *kovariant* in V – kovariant ist immer der Faktor mit dem $\vee$. In der Einstein-Notation steht der kontravariante Index oben und der kovariante unten; die Matrix aus dem übernächsten Absatz hat also Komponenten F^i_j . Die Basisvektoren tragen es genau umgekehrt, $w_i \otimes v^j$.

Gegeben $\mathbb{R}^n$ mit der Standardbasis und dem Standardskalarprodukt können wir Vektoren in Kovektoren verwandeln (und zurück) durch Transponieren: Der Spaltenvektor $v \in \mathbb{R}^n$ wird zum Zeilenvektor $v^T \in (\mathbb{R}^n)^\vee$, und die Matrixmultiplikation $v^T w = v \cdot w \in \mathbb{R}$ ist das Skalarprodukt. Weil die Standardbasis orthonormal ist, gilt hier $e_i^\vee = (e_i \cdot)$; in einer schiefen Basis stimmt das nicht.

Skalare, Vektoren und Matrizen als Tensoren. Gegeben einen Vektorraum V lässt sich das iterierte Tensorprodukt $V^{\otimes k} = V \otimes \dots \otimes V$ (k mal) betrachten, mit der Konvention $V^{\otimes 0} = \mathbb{R}$ (denn $V \otimes \mathbb{R} = V$). Eine Matrix $M \in \mathbb{R}^{n \times m}$ lässt sich auffassen als lineare Abbildung $F: \mathbb{R}^m \rightarrow \mathbb{R}^n$; die Spalten der Matrix sind $F(e_1), \dots, F(e_m)$, die Bilder der Basisvektoren. Für jeden Vektor $v \in \mathbb{R}^m$ ist stets

$$v = \sum_{i=1}^m (e_i \cdot v) e_i, \quad \text{also} \quad F(v) = \sum_{i=1}^m (e_i \cdot v) F(e_i), \quad (30)$$

und damit

$$F = \sum_{i=1}^m F(e_i)(e_i \cdot) = \sum_{i=1}^m F(e_i) \otimes e_i^\vee \in \mathbb{R}^n \otimes (\mathbb{R}^m)^\vee. \quad (31)$$

Jedes Element eines Tensorproduktraums wird *Tensor* genannt – also auch Skalare $x \in \mathbb{R} = V^{\otimes 0}$, Vektoren $v \in V = V^{\otimes 1}$ und lineare Abbildungen $F \in \text{Hom}(W, V) = V \otimes W^\vee$. Ein Skalar ist ein Tensor der *Stufe* 0, ein Vektor der Stufe 1, eine Matrix der Stufe 2; die Stufe zählt die Faktoren im Tensorprodukt und heißt in NumPy und PyTorch `ndim`. Im Englischen und in der Dokumentation von TensorFlow heißt dieselbe Zahl *rank*, weshalb man auch „Tensor vom Rang 2“ liest – beide Wörter sind gebräuchlich. Achtung, Kollision: Der *Rang einer Matrix* ist in der linearen Algebra etwas ganz anderes, nämlich die Dimension ihres Bildes. `torch.zeros(3,3).ndim` ist 2, `torch.linalg.matrix_rank(torch.zeros(3,3))` dagegen 0. Und was ist $T \in \mathbb{R}^n \otimes \mathbb{R}^m \otimes (\mathbb{R}^p)^\vee \otimes (\mathbb{R}^q)^\vee$ für ein Tensor? Eine bilineare Abbildung $\mathbb{R}^p \times \mathbb{R}^q \rightarrow \mathbb{R}^n \otimes \mathbb{R}^m$, zum Beispiel – oder eine lineare Abbildung $\mathbb{R}^p \otimes \mathbb{R}^q \rightarrow \mathbb{R}^n \otimes \mathbb{R}^m$ oder vier Indizes, je nach Lesart.

Obacht: lineare Abbildung $\neq$ Matrix. Jede Matrix lässt sich als lineare Abbildung interpretieren. Umgekehrt lässt sich, gegeben eine Wahl von Basen, jede lineare Abbildung als Matrix schreiben, die *darstellende Matrix*. Genauso hat der gleiche Vektor $v \in V$ verschiedene darstellende Vektoren $[v]_B \in \mathbb{R}^n$ für verschiedene Basen B . Zu einem Tensor $T \in U \otimes V \otimes W$ lässt sich ebenso eine Darstellung finden, wenn man Basen von U , V und W wählt: Sie hat zu jedem Tripel von Indizes (i, j, k) ein „Matrixelement“, nämlich den Koeffizienten von $u_i \otimes v_j \otimes w_k$. Das lässt sich dreidimensional darstellen, analog zur zweidimensionalen Darstellung für Matrizen.

Zurück zu NumPy & Co. In der Mathematik müssen wir uns zu einem Tensor merken, wie das Tensorprodukt zustande kam, um zu wissen, welche Variablen kovariant und welche kontravariant sind. Wählt man überall Basen, so verschwindet der Unterschied – man sieht nur noch die Koeffizienten. In NumPy arbeiten wir implizit mit Darstellungen in der Standardbasis von $\mathbb{R}^n$. Ist der Shape eines Arrays `a` z.B. `a.shape == (4, 10, 2)`, so bedeutet dies mathematisch gesehen

$$a \in \mathbb{R}^4 \otimes \mathbb{R}^{10} \otimes \mathbb{R}^2. \quad (32)$$

Quizfrage

Was macht `reshape`?

Antwort: Wegen $\mathbb{R}^n \otimes \mathbb{R}^m \cong \mathbb{R}^{nm}$ hat $\mathbb{R}^4 \otimes \mathbb{R}^{10}$ eine Basis der Form $e_i \otimes e_j$ mit insgesamt $4 \cdot 10 = 40$ Elementen, also ist `a.reshape(-1, 2) == a.reshape(40, 2)` derselbe Tensor, nur mit anderer Klammerung der Faktoren. `Reshape` ändert keine Daten, nur die Interpretation des Speichers.

Ausflug beendet.

Das Bild, das man sich für PyTorch merken sollte. Der Shape sagt nur, wie viele Zahlen in jede Richtung gehen, nicht, welche Richtung was bedeutet. Das muss man sich selbst merken, und genau an dieser Stelle entstehen die Fehler, die keinen Fehler werfen, sondern ohne Murren einen sinnlosen Output produzieren (der Autor hat einmal bei einem quadratischen Bild die beiden Achsen vertauscht und lange gesucht). Die Konvention, die fast überall in PyTorch gilt: Die *erste* Achse ist der Batch. Ein Tensor mit Shape `(32, 784)` sind 32 Datenpunkte mit je 784 Features, eine Zeile pro Datenpunkt. `(32, 28, 28)` sind 32 Bilder, also 32 Matrizen übereinandergestapelt; `(32, 1, 28, 28)` sind dieselben Bilder mit ausgeschriebener Kanalachse, `(32, 3, 28, 28)` mit drei Farbkanälen. Die Faltungsschichten im Kapitel „[Faltung und Convolutional Neural Networks](#)“ verlangen die Kanalachse, deshalb ist (N, C, H, W) die übliche Form für Bildbatches; solange keine Faltung folgt, lässt man den einen Kanal oft weg. `nn.Linear` und fast alle anderen Layer wirken auf die *letzte* Achse und lassen die Batch-Achse in Ruhe. Das „fast“ betrifft die rekurrenten Schichten: `nn.RNN`, `nn.LSTM`, `nn.GRU` und `nn.Transformer` erwarten voreingestellt `(seq, batch, feature)` und legen den Batch damit auf die *zweite* Achse; `batch_first=True` stellt die Regel wieder her. Wer einen Tensor vor sich hat, sollte als Erstes sagen können, welche Achse der Batch ist und was die übrigen Achsen bedeuten; `reshape` und `view` ändern nur die Klammerung, nicht die Bedeutung.

2.7.2 Übung

1. Welchen Shape hat das Ergebnis? (a) `torch.zeros(4,3) @ torch.zeros(3,7)`, (b) `x.sum(dim=0)` für `x` mit Shape `(8,5)`, (c) `x.unsqueeze(1)` für `x` mit Shape `(8,)`, (d) `torch.zeros(4,1) +`

`torch.zeros(1,6)`. *Tipp: Bei (d) hilft die Broadcasting-Regel: Achsen der Länge 1 werden gedehnt.*

2. Ein Batch von 32 Graustufenbildern der Größe 28×28 soll in eine Faltungsschicht. Welchen Shape hat er in der Konvention (N, C, H, W) , und welchen nach `nn.Flatten()`?

Antwortprobe: Das Produkt der Shape-Einträge ändert sich bei (2) nicht.

Weiterlesen

Deisenroth et al., *MML*, Kap. 2 (Linear algebra) für Basen, darstellende Matrizen und Basiswechsel. Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).

3 Lernen

3.1 Logistische Regression

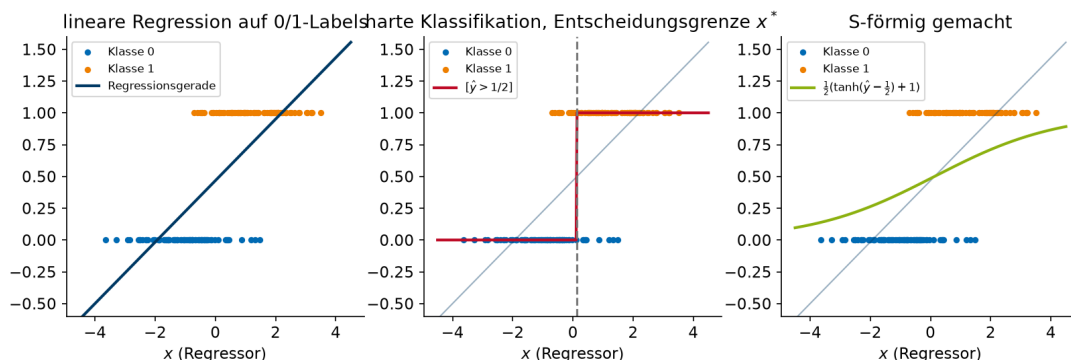
Im ersten Abschnitt basteln wir eine S-förmige Kurve aus dem tanh, haben aber keine Interpretation dafür. Danach bauen wir die S-Kurve von der Wahrscheinlichkeitsrechnung her auf, und dann fällt die Interpretation von selbst ab.

3.1.1 Lineare Regression zur Klassifikation

Wenn wir an Regression denken, dann oft an eine Regressionsgerade in einem x - y -Plot, bei dem auf der x -Achse die unabhängige Variable (der *Regressor*) und auf der y -Achse die abhängige Variable (der *Regressand*) aufgetragen ist. Bei der Klassifikation ist der Regressand die Klasse, $y \in \{0, 1\}$.

Schauen wir uns das Schritt für Schritt an, für einen eindimensionalen Regressor $x \in \mathbb{R}$ mit zwei Klassen, deren x -Werte sich überlappen:

1. Im eindimensionalen Plot liegen alle Punkte der Klasse 0 auf der Höhe $y = 0$ und alle Punkte der Klasse 1 auf $y = 1$.
2. Die lineare Regression fittet eine Gerade $\hat{y} = wx + b$ durch diese Punktwolke. Die Gerade nimmt Werte in ganz $\mathbb{R}$ an, auch weit außerhalb von $[0, 1]$.
3. **Harte Klassifikation:** Wir entscheiden für Klasse 1, wenn $\hat{y} > \frac{1}{2}$. Die Entscheidungsgrenze ist der Punkt x^* mit $w x^* + b = \frac{1}{2}$; die vorhergesagte Klasse als Funktion von x ist eine Sprungfunktion.
4. Für einen zweidimensionalen Regressor $x \in \mathbb{R}^2$ wird aus der Geraden eine Regressionshyperebene über der (x_1, x_2) -Ebene, und die Entscheidungsgrenze $\{w \cdot x + b = \frac{1}{2}\}$ ist eine Gerade in der (x_1, x_2) -Ebene – genau wie beim Perzeptron.



Lineare Regression zur Klassifikation, eindimensionaler Regressor: Regressionsgerade durch die 0/1-Daten, harte Klassifikation mit Entscheidungsgrenze x^* , und die tanh-Variante.

S-Förmigkeit herstellen

Die Sprungfunktion $[\hat{y} > \frac{1}{2}]$ ist unbefriedigend: Sie ist nicht differenzierbar, und sie verrät nicht, wie *sicher* die Entscheidung ist. Anstelle von $[\hat{y} > \frac{1}{2}]$ könnte man $[\tanh(\hat{y}) > 0]$ verwenden; tanh liegt zwischen -1 und +1. Um zwischen 0 und 1 zu landen, kann man z.B.

$$\frac{1}{2} (\tanh(\hat{y}) + 1) \quad (33)$$

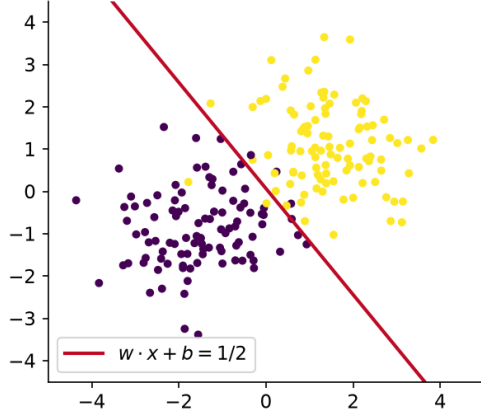
nutzen. Das liefert eine S-förmige Kurve, die glatt von 0 nach 1 übergeht. Aber: Die Interpretation einer Zahl zwischen 0 und 1 als *Wahrscheinlichkeit* ist damit noch unklar. Genau das klären die nächsten Abschnitte dieses Kapitels, über Odds und Logit.

3.1.2 Wahrscheinlichkeiten und Chancen (Odds)

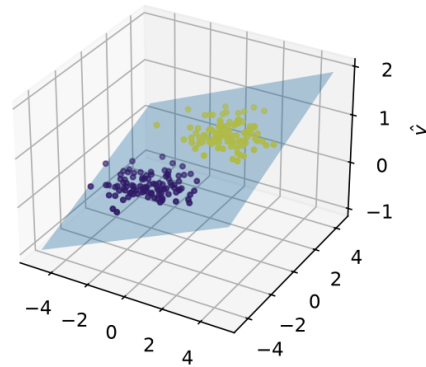
Chancen für ein Bernoulli-Experiment

Angenommen, wir haben ein Bernoulli-Experiment (Münzwurf) mit Parameter $p \in (0, 1)$ – zur Vereinfachung mit $p \neq 0$ und $p \neq 1$. Die **Wahrscheinlichkeit** (probability) für Erfolg ist $P(\text{Erfolg}) = p$. Die **Chancen** (odds) für Erfolg sind $p : (1 - p)$.

2D-Regressor: Entscheidungsgrenze ist eine Gerade



Regressionshyperebene



Zweidimensionaler Regressor: Die Entscheidungsgrenze ist eine Gerade, die Regressionsfunktion eine Ebene.

Beispiel: Einen Standardwürfel (D6) rollen und gucken, ob es eine 6 ist. Dann ist $P(D = 6) = \frac{1}{6} = p$ die Wahrscheinlichkeit, und $1 : 5 = \frac{1}{5}$ sind die Chancen für eine 6; $5 : 1 = 5$ sind die Chancen für eine Nicht-6. Als Formel:

$$\text{odds}(D = 6) = \frac{P(D = 6)}{P(D \neq 6)} = \frac{1/6}{5/6} = \frac{1}{5}. \quad (34)$$

Allgemein rechnet man von Wahrscheinlichkeiten zu Odds und zurück über

$$\text{odds}(p) = \frac{p}{1-p}, \quad p = \frac{\text{odds}}{1 + \text{odds}}. \quad (35)$$

Die Skala für Odds

Wahrscheinlichkeiten liegen immer in $[0, 1]$. Wie sieht die Skala für Odds aus?

- Wenn $p = \frac{1}{2}$, so ist $1 - p = \frac{1}{2}$, also $\text{odds}(p) = 1$.
- Wenn $0 < p < \frac{1}{2}$, so ist $1 > 1 - p > \frac{1}{2}$, also $0 < \text{odds}(p) = \frac{p}{1-p} < \frac{p}{1/2} = 2p < 1$. Damit ist $\text{odds}(p) \in (0, 1)$.
- Wenn $1 > p > \frac{1}{2}$, so ist $0 < 1 - p < \frac{1}{2}$, also $1 < 2p < \text{odds}(p) < \frac{1}{1-p} < \infty$. Damit ist $\text{odds}(p) \in (1, \infty)$.

Die Fälle $p < \frac{1}{2}$ und $p > \frac{1}{2}$ sind symmetrisch, d.h. Erfolg und Misserfolg sind äquivalent – aber $\text{odds}(p) \in (0, \infty)$ ist *nicht* symmetrisch: $(0, 1)$ ist beschränkt, $(1, \infty)$ ist unbeschränkt. Die Odds für eine 6 sind $\frac{1}{5}$, die Odds für eine Nicht-6 sind 5.

Beachte: Logarithmieren rettet die Symmetrie. Es ist

$$\ln\left(\frac{p}{1-p}\right) = \ln(p) - \ln(1-p), \quad (36)$$

und damit $\ln(1 : 5) = -\ln(5 : 1)$. Die Werte liegen in $(-\infty, +\infty)$.

Logistic Unit

Definition 3.1 (Logit). Für $p \in (0, 1)$ ist

$$\text{logit}(p) := \ln \text{odds}(p) = \ln \frac{p}{1-p} = \ln p - \ln(1-p). \quad (37)$$

Schon mal irgendwo gesehen? Im Data-Science-Skript haben wir den *Informationsgehalt* (die Überraschung) eines Ereignisses mit Wahrscheinlichkeit p als $-\log p$ definiert und die *Entropie* als erwartete Überraschung ([Informationsgehalt](#), [Entropie](#)). Der Logarithmus steht dort aus einem Grund, der uns noch oft nützen wird: Für unabhängige Ereignisse multiplizieren sich die Wahrscheinlichkeiten, und log macht

daraus eine Summe. Zwei Überraschungen addieren sich. Ein fairer Münzwurf ist 1 bit Überraschung, eine 6 beim Würfeln $\log_2 6 \approx 2,58$ bit, beides zusammen 3,58 bit.

Für das Bernoulli-Experiment ist die Entropie

$$H(p) = -p \ln p - (1-p) \ln(1-p), \quad (38)$$

und ihre Ableitung nach p ist

$$H'(p) = -\ln p - 1 + \ln(1-p) + 1 = -\text{logit}(p) \quad (39)$$

(im DS-Skript: [Die binäre Entropiefunktion](#), dort mit $\log_2$ und kleinem h). Interpretation: Bei Erhöhen von p ist die Abnahme an Informationsgewinn durch $\text{logit}(p)$ gegeben. Entropie misst man in bit ($\log_2$) oder nat ($\ln$), die Ableitung in *logit*. Zum Beispiel ist $H(0,5) = 1$ bit = 0,69 nat.

3.1.3 Logistische Sigmoidfunktion

Allgemeine S-Förmigkeit

Sigma $\approx$ S, Sigmoid $\approx$ S-förmig.

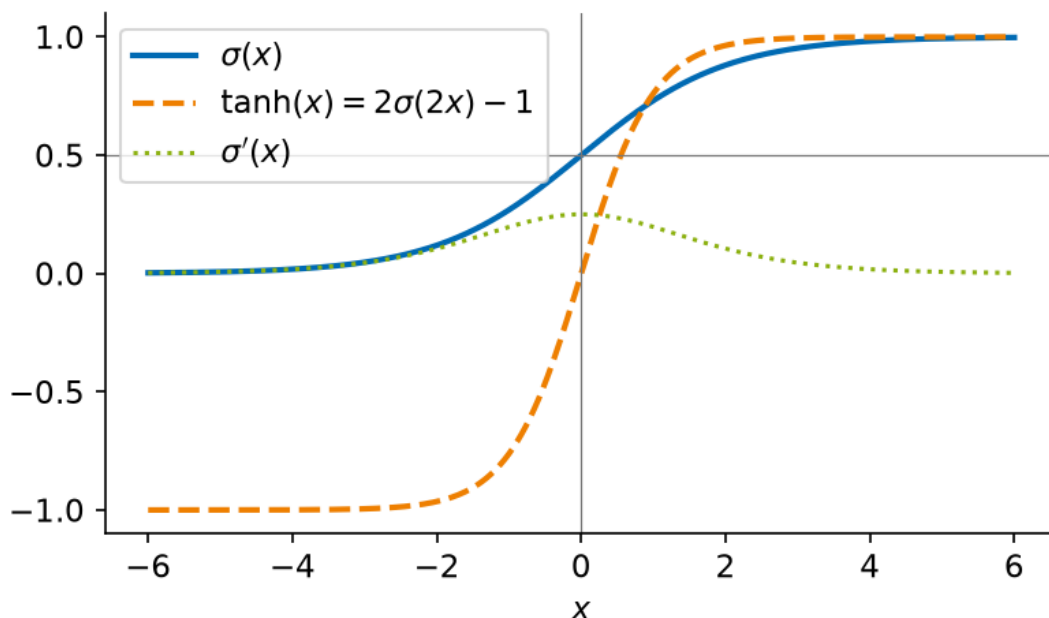
Definition 3.2 (Logistische Funktion). Die allgemeine logistische Funktion (aka allgemeine Sigmoidfunktion) mit Parametern M, k, μ ist gegeben als Funktion in x :

$$S(x; M, k, \mu) = \frac{M}{1 + e^{-k(x-\mu)}}. \quad (40)$$

Das Standardsigmoid ist

$$\sigma(x) = S(x; M = 1, k = 1, \mu = 0) = \frac{1}{1 + e^{-x}}. \quad (41)$$

M ist der Maximalwert (die obere Asymptote), k die Steilheit, μ die Position des Wendepunkts. Bei $\sigma(w \cdot x + b)$ übernimmt $\|w\|$ die Rolle von k , siehe [Kapitel „Neuron und Perzeptron“](#): Die Länge von w bestimmt, wie steil die Wahrscheinlichkeit mit dem Abstand zur Entscheidungsgrenze wächst. Das Standardsigmoid steigt monoton von 0 (für $x \rightarrow -\infty$) nach 1 (für $x \rightarrow +\infty$) und hat bei $x = 0$ den Wert $\sigma(0) = \frac{1}{2}$.



Das Standardsigmoid σ , der Tangens hyperbolicus als umskaliertes Sigmoid, und die Ableitung $\sigma' = \sigma(1 - \sigma)$ mit Maximum $\frac{1}{4}$ bei 0.

Symmetrie

Es gilt

$$\sigma(-x) = \frac{1}{1+e^x} = \frac{e^{-x}}{e^{-x}+1} = \frac{1+e^{-x}-1}{1+e^{-x}} = 1-\sigma(x), \quad (42)$$

also $\sigma(-x) = 1 - \sigma(x)$: Der Graph ist punktsymmetrisch um $(0, \frac{1}{2})$.

Andere Sigmoidfunktionen: Tangens hyperbolicus

Die Parameter M und k sind Skalierungsfaktoren. Möchte man die Skala des Standardsigmoids von $[0, 1]$ auf z.B. $[-1, 1]$ bringen, so nimmt man $2\sigma(x) - 1$, und mit $k = 2$:

$$2\sigma(2x) - 1 = \frac{2}{1+e^{-2x}} - 1 = \frac{1-e^{-2x}}{1+e^{-2x}} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \tanh(x). \quad (43)$$

Der Tangens hyperbolicus ist also nur ein umskaliertes Standardsigmoid, $\tanh(x) = 2\sigma(2x) - 1$.

Ableitung

Wir benötigen später die Ableitung der Sigmoidfunktion. Eventuell bekannt ist $\tanh'(x) = 1 - \tanh^2(x)$. Ähnlich gilt

$$\sigma'(x) = \sigma(x) (1 - \sigma(x)) = \sigma(x) \sigma(-x). \quad (44)$$

Der Beweis mit der Quotientenregel ist eine lohnende Übung. Bemerkenswert daran: Die Ableitung lässt sich aus dem Funktionswert berechnen, ohne e^{-x} noch einmal auszuwerten – das ist beim Training praktisch.

3.1.4 Sigmoid und Logit

Wir haben zwei Funktionen:

$$\sigma: \mathbb{R} \rightarrow (0, 1), \quad \text{logit}: (0, 1) \rightarrow \mathbb{R}. \quad (45)$$

Also lassen sich beide Verkettungen bilden, $\sigma \circ \text{logit}: (0, 1) \rightarrow (0, 1)$ und $\text{logit} \circ \sigma: \mathbb{R} \rightarrow \mathbb{R}$. Was haben die Funktionen miteinander zu tun?

Für $p \in (0, 1)$ ist

$$\sigma(\text{logit}(p)) = \frac{1}{1+e^{-\ln \frac{p}{1-p}}} = \frac{1}{1+\frac{1-p}{p}} = \frac{p}{p+1-p} = p. \quad (46)$$

Für $x \in \mathbb{R}$ ist – erst $\sigma(x)$ mit e^x erweitern, dann $\ln$ anwenden und $\ln(e^x) = x$ nutzen –

$$\text{logit}(\sigma(x)) = \ln \frac{\sigma(x)}{1-\sigma(x)} = \ln \frac{\frac{e^x}{e^x+1}}{\frac{1}{e^x+1}} = \ln e^x = x. \quad (47)$$

Wir haben gesehen: σ und logit sind Inverse zueinander,

$$\sigma \circ \text{logit} = \text{id}_{(0,1)}, \quad \text{logit} \circ \sigma = \text{id}_{\mathbb{R}}. \quad (48)$$

Darum nennt man σ als Inverse von logit auch **expit**.

Wie interpretieren wir das?

Gegeben einen Wert $x \in \mathbb{R}$ gibt $\sigma(x)$ einen Wert $p := \sigma(x) \in (0, 1)$, sodass $x = \text{logit}(p)$. Jeder Wert x lässt sich also als *Logit* auffassen – als logarithmierte Chance –, und $\sigma(x)$ gibt die Wahrscheinlichkeit des Bernoulli-Experiments an, zu der x der Logit-Wert ist. Das ist die Interpretation, die uns im ersten Abschnitt gefehlt hat: Eine Zahl zwischen 0 und 1, die aus σ kommt, ist eine Wahrscheinlichkeit, wenn die Eingabe ein Logit war.

3.1.5 Logistische Regression

Alle Zutaten sind bereit, jetzt wird gekocht. Die lineare Regression spuckt $\hat{y}$ -Werte aus, die in ganz $\mathbb{R}$ liegen. Wenn $y = 0$ und $y = 1$ zu den zwei Klassen korrespondieren, so fassen wir $\hat{y} = w \cdot x + b$ als Logit auf, und die Werte $\sigma(\hat{y})$ geben die Wahrscheinlichkeit an, in Klasse 1 zu liegen. Wenn $\sigma(\hat{y}) > \frac{1}{2}$, so entscheiden wir für Klasse 1 – und weil σ monoton mit $\sigma(0) = \frac{1}{2}$ ist, ist das dasselbe wie $\hat{y} > 0$.

Definition 3.3 (Logistische Regression). *Die logistische Regression ist das verallgemeinerte Perzeptron mit $\tau = \sigma$, also das Modell*

$$f(x) = \sigma(w \cdot x + b) = P(Y = 1 \mid X = x). \quad (49)$$

In Scikit-Learn wird in der logistischen Regression unterschieden zwischen `predict` und `predict_proba`. Den Wert $\hat{y} = w \cdot x + b$ selbst liefert `decision_function`, und damit ist

$$\text{decision_function}(x) = \hat{y}, \quad \text{predict_proba}(x) = (1 - \sigma(\hat{y}), \sigma(\hat{y})), \quad \text{predict}(x) = [\sigma(\hat{y}) > \tfrac{1}{2}]. \quad (50)$$

`predict_proba` gibt also **eine Spalte pro Klasse** zurück, in der Reihenfolge von `classes_`: bei Labels 0/1 zuerst $P(Y = 0)$, dann $P(Y = 1)$. Die Wahrscheinlichkeit für Klasse 1 steht in der zweiten Spalte. Beim Softmax weiter unten begegnet uns dieselbe Form für K Klassen.

Lineare Regression in PyTorch

In PyTorch macht man eine lineare Regression so:

```
import torch
import torch.nn as nn

class LinearRegression(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        return self.linear(x)
```

Ein Modell ist eine Unterklasse von `nn.Module`; im Konstruktor werden die Bausteine mit Parametern angelegt (hier eine einzige lineare Schicht `nn.Linear`, das ist $x \mapsto Wx + b$), und `forward` beschreibt die Inferenz. Das Training:

```
import numpy as np

x = np.linspace(0, 10, 50) # irgendein eindimensionaler Datensatz
y = 2.5 * x + 1.0 + np.random.normal(0, 1, 50) # verrauschte Gerade

model = LinearRegression(1, 1)
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

inputs = torch.from_numpy(x.reshape(-1, 1).astype("float32"))
labels = torch.from_numpy(y.reshape(-1, 1).astype("float32"))

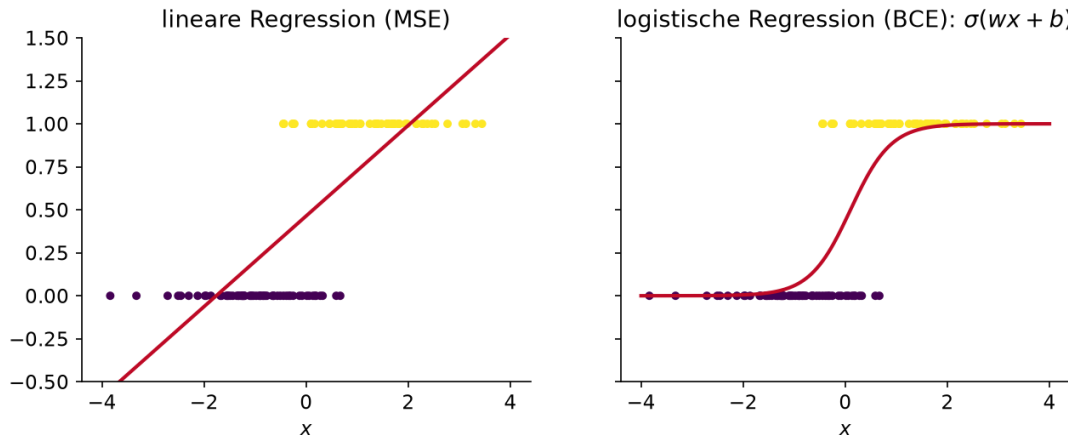
number_of_epochs = 100
losses = np.zeros(number_of_epochs, dtype="float32")

for epoch in range(number_of_epochs):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

```
losses[epoch] = loss.item()

y_pred = model(inputs).detach().numpy()
```

Die fünf Zeilen in der Schleife sind *die* Trainingsschleife von PyTorch, die uns in allen weiteren Kapiteln begleitet: Gradienten auf null setzen, Vorwärtsrechnung, Loss berechnen, Gradienten rückwärts berechnen (**backward**), Parameter anpassen (**step**). Was **criterion** (die Loss-Funktion, hier der mittlere quadratische Fehler MSE) und **optimizer** (hier stochastischer Gradientenabstieg SGD mit Lernrate 0,01) genau tun, klären die nächsten beiden Kapitel.



Dieselben 0/1-Daten, einmal mit linearer Regression (MSE-Loss) und einmal mit logistischer Regression (BCE-Loss) gefittet.

Logistische Regression in PyTorch

Für die logistische Regression ändern sich genau zwei Zeilen: Die Aktivierungsfunktion in **forward**

```
class LogisticRegression(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        return torch.sigmoid(self.linear(x)) # <---
```

und die Loss-Funktion:

```
model = LogisticRegression(1, 1)
criterion = nn.BCELoss() # <---
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

Der Rest der Trainingsschleife bleibt identisch. **BCELoss** ist die *binäre Kreuzentropie*: die erwartete Überraschung, wenn man mit den vorhergesagten Wahrscheinlichkeiten $(\sigma(\hat{y}), 1 - \sigma(\hat{y}))$ rechnet, die Wahrheit aber $(y, 1 - y)$ ist. Sie ist bis auf eine Konstante die Kullback-Leibler-Divergenz aus dem [DS-Skript](#) (dort $H(p\|r)$ geschrieben); den genauen Zusammenhang und woher diese Loss-Funktion *kommt*, klärt das nächste Kapitel.

3.1.6 Weg von der 2-Klassen-Gesellschaft: Softmax

Von Bernoulli zu Multinoulli

Von binomial (2 Klassen) zu multinomial (K Klassen): Anstelle eines einzelnen $\hat{y}$ -Werts der linearen Regression, der uns mit σ die Wahrscheinlichkeit für Klasse 1 angibt, benötigen wir nun einen Vektor der Länge K , der sich auf 1 aufsummiert. Für $K = 2$ erwarten wir den Vektor $(\sigma(\hat{y}), 1 - \sigma(\hat{y}))$.

Wenn wir die Klasse **one-hot** kodieren, also Vektoren der Länge K mit einer 1 an Stelle k und sonst Nullen als Labels verwenden, dann liefert multinomiale lineare Regression – eine Matrix $W \in \mathbb{R}^{K \times n}$ statt

eines Vektors w – Vorhersagevektoren $v = Wx + b \in \mathbb{R}^K$. Die Entscheidung für Klasse $k \in \{1, \dots, K\}$ fällt, wenn der Eintrag des Vektors an Index k am größten ist, $\operatorname{argmax}_j v_j = k$. Gesucht ist, analog zu σ , ein „soft argmax“.

Definition 3.4 (Softmax). Die Softmax-Funktion $\operatorname{sm}: \mathbb{R}^K \rightarrow \Delta^K$ ist komponentenweise gegeben durch

$$\operatorname{sm}_j(v) = \frac{e^{v_j}}{\sum_{k=1}^K e^{v_k}}. \quad (51)$$

Dabei ist Δ^K die Teilmenge von $[0, 1]^K$ mit $\sum_j v_j = 1$, der Wahrscheinlichkeitssimplex.

Man darf sich das zweistufig vorstellen: Im ersten Schritt wird der Vektor v komponentenweise exponenziert (alle Einträge werden positiv, und die Reihenfolge bleibt erhalten), im zweiten Schritt wird der Vektor normiert, sodass die Summe 1 ist. Für den Fall $K = 2$ mit $v = (x, y)$ können wir leicht einsehen, dass

$$\operatorname{sm}_1(v) = \frac{e^x}{e^x + e^y} = \frac{1}{1 + e^{y-x}} = \sigma(x - y), \quad (52)$$

Softmax verallgemeinert also tatsächlich das Sigmoid, wobei nur die *Differenz* der Logits zählt. Obacht beim Vergleich mit σ : In der Softmax-Formel steht *kein* Minus im Exponenten. Das Minus in $\sigma(x) = 1/(1 + e^{-x})$ entsteht erst beim Kürzen für $K = 2$, wie eben gerechnet. Die Probe, dass $\operatorname{sm}(v)$ wirklich eine Wahrscheinlichkeitsverteilung ist: Alle Einträge sind positiv, und die Summe ist $\sum_j e^{v_j} / \sum_k e^{v_k} = 1$.

Obacht!

Nicht jedes Multiklassen-Klassifikationsproblem schreit nach dem Softmax. Softmax ist dann sinnvoll, wenn argmax sinnvoll wäre, also wenn sich die Klassen gegenseitig ausschließen (ein Bild zeigt *entweder* eine Katze *oder* einen Hund). Sonst – z.B. bei Tags, von denen mehrere gleichzeitig zutreffen können – nimmt man paarweise binomiale Klassifikation mit Sigmoid, ein σ pro Klasse.

Obacht: e^x und $\ln$ am Computer

In `float32` ist e^{89} bereits `inf`, und $\sigma(-100) = 1/(1 + e^{100})$ wird zu `1/inf = 0` gerundet. Wer dann die Kreuzentropie selbst ausschreibt, berechnet $\ln 0 = -\infty$ und $0 \cdot (-\infty) = \text{nan}$; der Loss ist `nan`, und ab dem nächsten Schritt sind es alle Gewichte auch. Ähnlich beim Softmax: e^{v_j} läuft über, sobald ein Logit größer als etwa 88 ist.

Abhilfe ist immer dieselbe: nie erst e^x und dann $\ln$ berechnen, sondern beides algebraisch zusammenfassen. Zum Beispiel ist $\ln \sigma(z) = -\ln(1 + e^{-z})$, und für $z < 0$ schreibt man das als $z - \ln(1 + e^z)$, dann ist der Exponent nie groß. Beim Softmax darf man von allen v_k das Maximum abziehen, ohne das Ergebnis zu ändern (*log-sum-exp-Trick*). PyTorch hat das eingebaut: `nn.BCEWithLogitsLoss` und `nn.CrossEntropyLoss` nehmen die *Logits* entgegen, nicht die Wahrscheinlichkeiten, und rechnen stabil. (`nn.BCELoss` auf Wahrscheinlichkeiten fängt $\ln 0$ mit einem Trick ab, indem es den Logarithmus bei -100 abschneidet. Das verhindert `nan`, ist aber ein Pflaster, keine Lösung.) Praktisch heißt das: Wenn Ihr Loss `nan` wird, ist meist das die Ursache, nicht ein Fehler in Ihrer Trainingsschleife.

3.1.7 Übung

1. Zeigen Sie $\sigma(-x) = 1 - \sigma(x)$. *Tipp: Erweitern Sie $\sigma(-x) = \frac{1}{1+e^x}$ mit e^{-x} .*
2. Zeigen Sie $\sigma'(x) = \sigma(x)\sigma(-x)$. *Tipp: Quotientenregel, dann mit $1/\sigma(-x)$ multiplizieren und so lange umformen, bis die Definition von $\sigma(x)$ dasteht. Mit Teil 1 ist das dasselbe wie die aus der Literatur bekannte Form $\sigma(x)(1 - \sigma(x))$.*
3. Wie groß ist $\sigma'(0)$? Und wie groß sind die Odds bei $p = 0,5$?

Antwortprobe: Die Odds sind 1. Die andere Zahl ist nach Teil 2 ein Produkt aus zwei gleichen Faktoren, also eine Quadratzahl; steht bei Ihnen $\frac{1}{2}$, haben Sie $\sigma(0)$ statt $\sigma'(0)$ ausgerechnet.

Weiterlesen

Prince, *UDL*, Kap. 5 (Loss functions), dort die logistische Regression als Maximum-Likelihood-Modell; Deisenroth et al., *MML*, Kap. 6 (Probability). Beide Bücher sind frei verfügbar, siehe [Über dieses Skript](#).

3.2 Loss-Funktionen und Maximum Likelihood

Dieses Kapitel klärt, *was* beim Training minimiert wird und warum. Das nächste Kapitel klärt, *wie*.

3.2.1 Training eines einzelnen Neurons: Was bisher geschah

Wir haben zwei Modelle für ein einzelnes Neuron gesehen, mit gleicher Struktur, aber sehr unterschiedlich beschriebenen Training:

	Perzeptron	Logistische Regression
Inferenz	$[\langle w, x \rangle > 0] \in \{0, 1\}$	$\sigma(\langle w, x \rangle) \in (0, 1)$
Aktivierungsfunktion	„sign“ (Vorzeichen)	σ (logistische Sigmoidfunktion)
Training	Update-Regel $w_{\text{neu}} = w_{\text{alt}} + \rho(y_{\text{true}} - f(x))x$	naiv: finde w , sodass der MSE der linearen Regression minimal wird

Beim Perzeptron kann jedes einzelne neue Paar (x, y) zum Weiterlernen benutzt werden; bei der logistischen Regression wird ein Fehler minimiert. Zwei Fragen drängen sich auf: Wie lernt man bei der logistischen Regression aus einem *einzelnen* neuen Paar (x, y) etwas dazu? Und wird beim Perzeptron eigentlich auch etwas maximiert oder minimiert?

Neue, einheitliche Perspektive

Die Antwort auf beide Fragen: Jedes Training ist die Minimierung eines **Fehlerterms**, der **Loss-Funktion** $L(x, y; w)$, und die Update-Regel folgt aus der Loss-Funktion. Für die logistische Regression ist der passende Fehlerterm die **binäre Kreuzentropie** (binary cross entropy, BCE):

$$L_{\text{BCE}}(x, y; w) = -\frac{1}{N} \sum_{i=1}^N \left(y_i \log p_i + (1-y_i) \log(1-p_i) \right), \quad p_i = P(x_i \text{ in Klasse 1}) = \sigma(\langle w, x_i \rangle) = \frac{e^{\langle w, x_i \rangle}}{1 + e^{\langle w, x_i \rangle}}. \quad (53)$$

Für das Perzeptron (mit Labels $y_i \in \{-1, +1\}$ statt $\{0, 1\}$, das macht die Formeln kürzer) ist ein erster Kandidat der „**harte**“ **Perceptron Loss**

$$L_{\text{Perc}}^{\text{hart}}(x, y; w) = \sum_{i=1}^N \max(0, -y_i \text{sign}(\langle w, x_i \rangle)), \quad (54)$$

der einfach die falsch klassifizierten Samples zählt. Warum dieser Kandidat für den Gradientenabstieg untauglich ist – er ist stückweise konstant – und wie die *weiche* Fassung ohne sign daraus die Perzeptronregel macht, steht im [nächsten Kapitel](#); dort leiten wir auch die Update-Regel zur BCE her.

Woher kommt die Kreuzentropie? Maximum Likelihood

Die logistische Regression behauptet: $P(Y = 1 \mid X = x) = p(x) := \sigma(\langle w, x \rangle)$. Für einen Datenpunkt (x, y) mit $y \in \{0, 1\}$ ist die Wahrscheinlichkeit des *beobachteten* Labels unter dieser Behauptung

$$P(Y = y \mid X = x) = \begin{cases} p(x) & y = 1 \\ 1 - p(x) & y = 0 \end{cases} = p(x)^y (1 - p(x))^{1-y}, \quad (55)$$

wobei die kompakte Schreibweise rechts funktioniert, weil einer der beiden Exponenten 0 ist und $a^0 = 1$. Für N unabhängig gezogene Datenpunkte multiplizieren sich die Wahrscheinlichkeiten. Die **Likelihood** der Gewichte w ist

$$\mathcal{L}(w) = \prod_{i=1}^N p(x_i)^{y_i} (1 - p(x_i))^{1-y_i}. \quad (56)$$

Das Prinzip der **Maximum-Likelihood-Schätzung** (**DS-Skript**): Wähle die Parameter, unter denen die beobachteten Daten am wahrscheinlichsten sind. Ein Produkt aus N Faktoren zwischen 0 und 1 ist unhandlich und numerisch schnell 0; der Logarithmus macht daraus eine Summe, ohne die Lage des Maximums zu ändern, denn $\ln$ ist streng monoton wachsend:

$$\ln \mathcal{L}(w) = \sum_{i=1}^N \left(y_i \ln p(x_i) + (1 - y_i) \ln(1 - p(x_i)) \right). \quad (57)$$

Maximieren der Log-Likelihood ist Minimieren der negativen Log-Likelihood, und die ist bis auf den Faktor $\frac{1}{N}$ genau L_{BCE} . Die binäre Kreuzentropie ist also keine willkürliche Wahl: Sie ist *die* Loss-Funktion, die aus „das Modell sagt Wahrscheinlichkeiten voraus, und wir glauben den Daten“ folgt. Dasselbe Argument mit Gauß-Rauschen $Y = \langle w, X \rangle + \varepsilon$, $\varepsilon \sim \mathcal{N}(0, s^2)$, liefert den MSE als Loss der linearen Regression (Übung: In der Gauß-Dichte ist ein quadratischer Term plus Konstante).

Example 3.1 (Maximum Likelihood von Hand). *Drei Münzwürfe ergeben Kopf, Kopf, Zahl. Ein Modell ohne Eingabe x , nur mit einem Parameter $p = P(\text{Kopf})$, hat Likelihood $\mathcal{L}(p) = p \cdot p \cdot (1 - p)$ und Log-Likelihood $2 \ln p + \ln(1 - p)$. Ableiten und null setzen:*

$$\frac{2}{p} - \frac{1}{1-p} = 0 \iff 2(1-p) = p \iff p = \frac{2}{3}. \quad (58)$$

*Das ist der Anteil der Köpfe, und zugleich der Wert, an dem die binäre Kreuzentropie dieser drei Datenpunkte (mit konstanter Vorhersage p) minimal ist. Im DS-Skript steht dieselbe Rechnung ohne Logarithmus (*Welches p passt am besten zu den Daten?*); vergleichen Sie, welche Variante angenehmer ist.*

3.2.2 Optimieren mit Fehlerterm

Wenn $L(x, y; w) \in \mathbb{R}$ den Fehler eines Modells mit Gewichtsvektor w angibt, dann suchen wir w , sodass $L(w) := L(x, y; w)$ minimal wird. Die Daten x, y sind dabei fest, variabel ist w .

Ableiten nach den Gewichten, nicht nach den Daten

In der Schule wird nach x abgeleitet. Hier ist x eine Konstante (der Datenpunkt) und w die Variable: Wir fragen, wie sich der Fehler ändert, wenn wir *am Modell* drehen, nicht an den Daten. Beides wird vorkommen, denn bei mehreren Schichten ist die Eingabe einer Schicht die Ausgabe der vorigen, und dann brauchen wir auch die Ableitung nach der *Eingabe*, um den Fehler an die vorige Schicht weiterzureichen. Beide Ableitungen desselben Ausdrucks nebeneinander:

Ausdruck	Ableitung nach w	Ableitung nach x
$\langle w, x \rangle$	x	w
$\sigma(\langle w, x \rangle)$	$\sigma'(\langle w, x \rangle) x$	$\sigma'(\langle w, x \rangle) w$
$(y - \langle w, x \rangle)^2$	$-2(y - \langle w, x \rangle) x$	$-2(y - \langle w, x \rangle) w$

Die Symmetrie ist kein Zufall: $\langle w, x \rangle$ ist in beiden Argumenten linear. Für Backpropagation heißt das: Wer $\frac{\partial}{\partial w}$ kann, kann auch $\frac{\partial}{\partial x}$, man tauscht nur die Rollen. Und in der Klausur: immer dazuschreiben, *wonach* abgeleitet wird.

Exakt, wie in der Schule: Extremstellen von L bestimmen.

1. L' bestimmen,
2. $L'(w) = 0$ nach w umstellen,
3. Formel für w ausrechnen (aus den Daten x, y),
4. Kandidaten für w ausprobieren – liegt ein Minimum vor?

Für die lineare Regression mit MSE klappt das (die **Normalgleichungen** aus Data Science). Schaffen wir das immer? Nein: Für die logistische Regression hat $L'(w) = 0$ keine geschlossene Lösung, und für tiefe Netze erst recht nicht.

Approximativ, ganz dumm:

1. Zufällig ein w auswählen,
2. $L(w)$ ausrechnen, vergleichen mit dem bisher niedrigsten Wert,

3. Größer? Pech gehabt, weitermachen. Kleiner? Besseres w gefunden!
4. Jetzt ggf. aufhören, wenn w gut genug ist.

Approximativ, clever: Wie eben, aber im ersten Schritt w nicht zufällig, sondern *in der Nähe der bisher besten Schätzung* auswählen. Das ist eine lokale Suche, und die Frage ist nur noch, in welche Richtung man von der besten Schätzung aus gehen soll. Das ist der Gradientenabstieg, Thema des nächsten Kapitels.

Obacht: Bei linear trennbaren Daten hört die logistische Regression nie auf

Zwei Datenpunkte in $\mathbb{R}$: $x = -1$ mit $y = 0$ und $x = +1$ mit $y = 1$, Modell $\sigma(wx)$ ohne Bias. Die BCE ist

$$L(w) = -\ln(1 - \sigma(-w)) - \ln \sigma(w) = -2 \ln \sigma(w) \quad (59)$$

(wegen $1 - \sigma(-w) = \sigma(w)$). Diese Funktion fällt streng monoton in w und nähert sich 0 für $w \rightarrow \infty$, erreicht es aber nie: Es gibt kein Minimum. Jedes Verfahren, das L verkleinert, schiebt w immer weiter, weil ein steileres Sigmoid (größeres $\|w\|$, siehe [Kapitel „Neuron und Perzeptron“](#)) den Datenpunkten immer mehr Wahrscheinlichkeit gibt. In der Praxis sieht das so aus: Der Loss sinkt und sinkt, die Gewichte wachsen, die Vorhersagen werden „sicherer“, ohne dass sich die Trenngerade noch bewegt. Abhilfe: ein Strafterm für $\|w\|$ (*Regularisierung*, wie bei der Ridge-Regression im [DS-Skript](#); in PyTorch `weight_decay` im Optimierer) oder einfach rechtzeitig aufhören.

Umgekehrt ist das der Vorteil gegenüber dem Perzeptron: Bei *nicht* trennbaren Daten hat die BCE ein Minimum, und die logistische Regression liefert eine sinnvolle Antwort, wo die Perzeptronregel nie terminiert.

3.2.3 Take-Home Message

Training heißt: eine Loss-Funktion $L(w)$ in den Gewichten minimieren, die Daten sind dabei Konstanten. Die binäre Kreuzentropie ist die negative Log-Likelihood der logistischen Regression, der MSE die der linearen Regression mit Gauß-Rauschen. Exakt lösen kann man das nur in Ausnahmefällen; sonst sucht man lokal, und dafür braucht man die Richtung, in der L am schnellsten fällt.

3.2.4 Übung

Seien q (die wahren Labels) und p (die Vorhersage) zwei Verteilungen auf $\{0, 1\}$. Dann sind

$$BCE(q, p) = -q(0) \log p(0) - q(1) \log p(1), \quad H(q) = -q(0) \log q(0) - q(1) \log q(1) \quad (60)$$

die binäre Kreuzentropie und die Entropie von q , und

$$D(q||p) = q(0) \log \frac{q(0)}{p(0)} + q(1) \log \frac{q(1)}{p(1)} \quad (61)$$

ist die Kullback-Leibler-Divergenz (im [DS-Skript](#) als $H(q||p)$ geschrieben).

1. Sei $q(0) = 0$ und $q(1) = 1$ (das wahre Label ist 1), dazu $p(0) = 0,2$ und $p(1) = 0,8$. Berechnen Sie $BCE(q, p)$ und $H(q)$. *Tipp: $0 \log 0$ wird als 0 gelesen. $H(q)$ ist hier besonders einfach.*
2. Zeigen Sie allgemein $D(q||p) = BCE(q, p) - H(q)$. *Tipp: Schreiben Sie den Logarithmus des Quotienten als Differenz von Logarithmen und sortieren Sie nach p und q .*
3. Warum darf man beim Training statt $D(q||p)$ einfach $BCE(q, p)$ minimieren?

Antwortprobe: In Teil 1 muss $D(q||p) = BCE(q, p)$ herauskommen.

Weiterlesen

Prince, *UDL*, Kap. 5 (Loss functions: Maximum Likelihood als Rezept für alle Loss-Funktionen); Deisenroth et al., *MMML*, Abschnitt 8.3 (Parameter estimation) und Kap. 9 (Linear regression). Beide Bücher sind frei verfügbar, siehe [Über dieses Skript](#).

3.3 Gradientenabstieg

Im [letzten Kapitel](#) blieb die Frage offen, in welche Richtung man von der bisher besten Schätzung w aus gehen soll, damit die Loss-Funktion kleiner wird. Die Antwort ist der Gradient, und um ihn für verkettete Funktionen auszurechnen, brauchen wir zuerst die Kettenregel.

3.3.1 Kurze Erinnerung an die Kettenregel

Für das Training werden wir Ableitungen von verketteten Funktionen brauchen, deshalb die Kettenregel in der Form, die wir benötigen: für Funktionen zwischen Vektorräumen.

Für eine Funktion $g: \mathbb{R}^n \rightarrow \mathbb{R}^m$ an einem Punkt x_0 ist die Ableitung $(Dg)(x_0)$ – eine $m \times n$ -Matrix, die Jacobi-Matrix – die beste lineare Approximation, also für x_1 nah bei x_0 :

$$g(x_1) \approx g(x_0) + (Dg)(x_0)(x_1 - x_0). \quad (62)$$

Gegeben eine weitere Funktion $f: \mathbb{R}^m \rightarrow \mathbb{R}^k$, ebenso

$$f(y_1) \approx f(y_0) + (Df)(y_0)(y_1 - y_0). \quad (63)$$

Im Spezialfall $y_0 = g(x_0)$, $y_1 = g(x_1)$ setzen wir ein:

$$\begin{aligned} f(g(x_1)) &\approx f(g(x_0)) + (Df)(g(x_0))(g(x_1) - g(x_0)) \\ &\approx f(g(x_0)) + (Df)(g(x_0))(g(x_0) + (Dg)(x_0)(x_1 - x_0) - g(x_0)) \\ &= f(g(x_0)) + (Df)(g(x_0))(Dg)(x_0)(x_1 - x_0). \end{aligned} \quad (64)$$

Damit ist $(Df)(g(x_0)) \cdot (Dg)(x_0)$ die beste lineare Approximation von $f \circ g$ bei x_0 . In anderer Notation der gleiche Sachverhalt:

$$D(f \circ g) = ((Df) \circ g) \cdot Dg. \quad (65)$$

Die Ableitung einer Verkettung ist das Matrixprodukt der Ableitungen, ausgewertet an den richtigen Stellen. Das ist bereits die halbe Backpropagation.

Summe über alle Pfade. Die Matrixform versteckt einen Fall, der beim Rechnen von Hand oft schiefgeht: Wenn eine Variable auf *mehreren Wegen* in das Ergebnis eingeht, addieren sich die Beiträge. Für $L = u \cdot v$ mit $u = g(w)$ und $v = h(w)$ ist

$$\frac{dL}{dw} = \frac{\partial L}{\partial u} g'(w) + \frac{\partial L}{\partial v} h'(w) = v g'(w) + u h'(w). \quad (66)$$

Kleinstes Beispiel: $u = w$ und $v = w$, also $L = w^2$. Wer nur einem Pfad folgt, erhält w ; richtig ist $w + w = 2w$. In einem Netz geht jedes Gewicht der ersten Schicht über *alle* Neuronen der zweiten Schicht in den Loss ein. Backpropagation ist im Kern die Buchhaltung dieser Summe, und die Matrixmultiplikation $(Df)(Dg)$ erledigt sie automatisch: Ein Matrixprodukt *ist* eine Summe über Zwischenindizes.

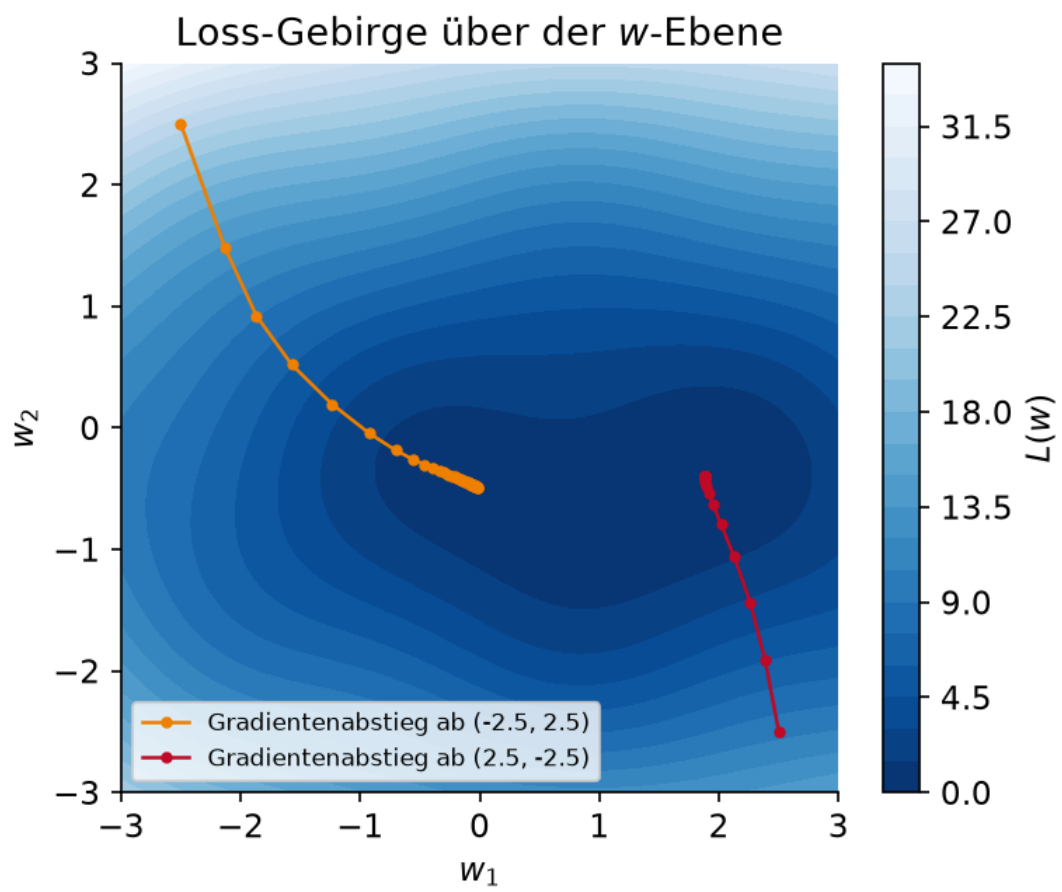
Numerische Probe. Ableitungen kann man nachprüfen, ohne sie zu kennen: Für kleines h ist

$$\frac{L(w+h) - L(w)}{h} \approx L'(w). \quad (67)$$

Beispiel: $L(w) = (1 - 2w)^2$, also $L'(w) = -4(1 - 2w)$ und $L'(1) = 4$. Nachrechnen: $L(1) = 1$, $L(1,01) = (1 - 2,02)^2 = 1,0404$, Differenzenquotient $0,0404/0,01 = 4,04$. Diese Probe (*gradient check*) ist der wichtigste Handgriff, wenn eine von Hand hergeleitete Ableitung oder ein selbst geschriebenes **backward** nicht stimmen will. PyTorch hat sie eingebaut: `torch.autograd.gradcheck`.

3.3.2 Gradienten

Wir stellen uns für den Moment vor, dass $w \in \mathbb{R}^2$ ist (also die Daten zwei Features haben). Wir stellen uns die w -Ebene vor und darauf ein $L(w)$ -Gebirge. An jedem Punkt die Frage: Wohin sollen wir gehen, wenn wir runter wollen? Antwort: In die Richtung, wo $L(w)$ am schnellsten kleiner wird, also „nach unten“ – das heißt entlang des negativen **Gradienten** $\text{grad}(L)(w) = L'(w) \in \mathbb{R}^2$.



Ein (erfundenes) Loss-Gebirge über der w -Ebene mit zwei Gradientenabstiegspfaden von verschiedenen Startpunkten. Beide landen im selben Tal – das ist nicht immer so.

Gradienten konkret

Betrachten wir den MSE für eine lineare Regression, mit der Notation $x_j^{(i)}$ für Feature j von Datenpunkt i aus dem [Kapitel „Was ist Deep Learning?“](#):

$$L(w) = \frac{1}{N} \sum_{i=1}^N (y_i - \langle w, x^{(i)} \rangle)^2. \quad (68)$$

Halte zunächst die Komponente w_2 fest und variiere w_1 . Das gibt eine Funktion in einer Variablen, $w_1 \mapsto L(w_1, w_2)$, und ihre Ableitung gibt die Änderung von L abhängig von w_1 an, also die Änderung in Richtung der ersten Koordinate der w -Ebene: „Wenn ich einen (infinitesimalen) Schritt nach Osten gehe, wie stark ändert sich L dann?“ Danach w_1 festhalten und w_2 variieren: ein Schritt nach Norden.

Diese beiden Ableitungen heißen besser **partielle Ableitungen** $\frac{\partial}{\partial w_1} L(w)$ und $\frac{\partial}{\partial w_2} L(w)$, und der Gradient ist der Vektor aus beiden:

$$L'(w) = \left(\frac{\partial L}{\partial w_1}(w), \frac{\partial L}{\partial w_2}(w) \right) = \frac{\partial L}{\partial w_1}(w) e_1 + \frac{\partial L}{\partial w_2}(w) e_2. \quad (69)$$

Nicht verwechseln: die Ableitung L' (eine Funktion, die jedem w einen Vektor zuordnet) und $L'(w)$, der Gradientenvektor an einem Punkt w .

Richtungsableitung. Die Änderung in Richtung e_1 ist $\langle L'(w), e_1 \rangle = \frac{\partial L}{\partial w_1}(w)$. Die Änderung in Richtung eines beliebigen Einheitsvektors v ist

$$\langle L'(w), v \rangle = \frac{\partial L}{\partial w_1}(w) v_1 + \frac{\partial L}{\partial w_2}(w) v_2. \quad (70)$$

In welche Richtung v ist die Änderung maximal? Das Skalarprodukt $\langle L'(w), v \rangle$ ist $\|L'(w)\| \cos \alpha$ mit dem Winkel α zwischen v und dem Gradienten, und der Kosinus ist maximal 1, wenn v in Richtung $L'(w)$ zeigt, und minimal -1 in der Gegenrichtung. Der Gradient zeigt bergauf, der negative Gradient bergab.

Vertiefung: dasselbe mit Cauchy-Schwarz

Ohne Winkel: Nach der Cauchy-Schwarz-Ungleichung ist $\langle L'(w), v \rangle \leq \|L'(w)\| \|v\|$ mit Gleichheit genau dann, wenn v ein nichtnegatives Vielfaches von $L'(w)$ ist. Für Einheitsvektoren v ist die rechte Seite $\|L'(w)\|$, und sie wird genau in Richtung $v = L'(w)/\|L'(w)\|$ erreicht. Das Argument funktioniert in jeder Dimension, auch dort, wo man sich keinen Winkel mehr vorstellen kann.

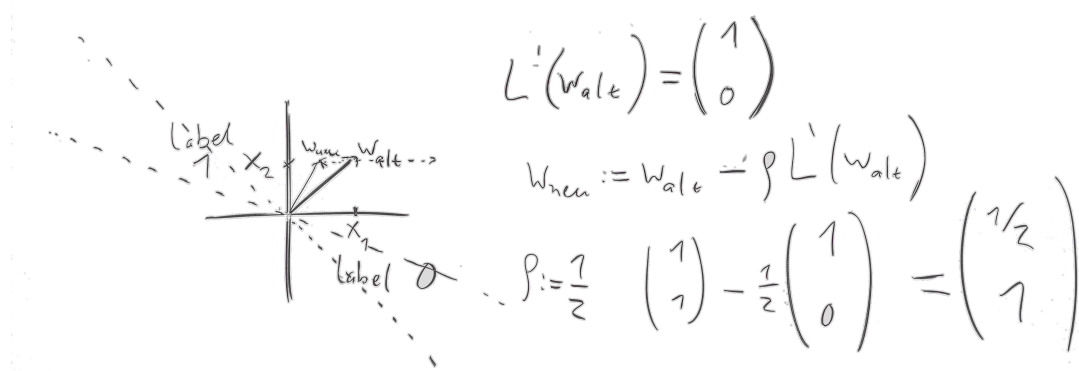
Endlich ausrechnen. Mit der Kettenregel:

$$\frac{\partial L}{\partial w_1}(w) = \frac{1}{N} \sum_{i=1}^N 2 (y_i - w_1 x_1^{(i)} - w_2 x_2^{(i)}) (-x_1^{(i)}), \quad \frac{\partial L}{\partial w_2}(w) = \frac{1}{N} \sum_{i=1}^N 2 (y_i - w_1 x_1^{(i)} - w_2 x_2^{(i)}) (-x_2^{(i)}). \quad (71)$$

Example 3.2 (Ein Gradientenschritt von Hand). *Fiktive Daten:* $x^{(1)} = (1, 0)$, $y_1 = 0$, $x^{(2)} = (0, 1)$, $y_2 = 1$, also $N = 2$, und $w_{alt} = (1, 1)$. Dann ist (der Faktor $\frac{2}{N} = 1$ fällt weg)

$$\begin{aligned} \frac{\partial L}{\partial w_1}(w_{alt}) &= \sum_{i=1}^2 (y_i - x_1^{(i)} - x_2^{(i)}) (-x_1^{(i)}) = (0 - 1)(-1) + (1 - 1)(-0) = 1, \\ \frac{\partial L}{\partial w_2}(w_{alt}) &= \sum_{i=1}^2 (y_i - x_1^{(i)} - x_2^{(i)}) (-x_2^{(i)}) = (0 - 1)(-0) + (1 - 1)(-1) = 0, \end{aligned} \quad (72)$$

also $L'(w_{alt}) = (1, 0)$. Der Update-Schritt $w_{neu} := w_{alt} - \rho L'(w_{alt})$ mit z.B. $\rho = \frac{1}{2}$ liefert $w_{neu} = (1, 1) - (\frac{1}{2}, 0) = (\frac{1}{2}, 1)$. Probe: Vorher war $\langle w, x^{(1)} \rangle = 1$ bei $y_1 = 0$, nachher $\frac{1}{2}$ – der Fehler auf dem ersten Sample ist gesunken, der auf dem zweiten unverändert. Das Minus vor dem Gradienten ist der ganze Unterschied zwischen descent und ascent: Mit Plus liefe man bergauf.



Tafelbild zum Gradientenschritt von Hand (WiSe 2025/26).

3.3.3 Der Algorithmus

Damit haben wir die sehr clevere Variante der approximativen Minimierung:

1. Sehr clever ein w auswählen, in der Nähe der bisher besten Schätzung:

$$w_{\text{neu}} := w_{\text{alt}} - \rho L'(w_{\text{alt}}). \quad (73)$$

2. $L(w)$ ausrechnen, vergleichen mit dem bisher niedrigsten Wert,
3. Größer? Pech gehabt, weitermachen (ggf. mit kleinerem ρ). Kleiner? Besseres w gefunden!
4. Jetzt ggf. aufhören, wenn w gut genug ist.

Das ist der **Gradientenabstieg**, und ρ ist die **Lernrate** – dieselbe, die wir beim Perzeptron schon hatten. Zu klein, und man braucht ewig; zu groß, und man springt über das Tal hinweg.

Wie groß „zu groß“ ist, hängt von den Daten ab, und zwar von jedem Feature einzeln. Für den MSE ist $\frac{\partial L}{\partial w_j} = \frac{1}{N} \sum_i 2(y_i - \langle w, x^{(i)} \rangle)(-x_j^{(i)})$, also proportional zu den Werten von Feature j . Hat Feature 1 Werte um 1 und Feature 2 Werte um 1000 (Körpergröße in Metern, Einkommen in Euro), so ist die zweite Gradientenkomponente etwa tausendmal größer als die erste, und die Krümmung des Loss-Gebirges in w_2 -Richtung ist eine Million mal größer: Das Tal ist eine lange, schmale Schlucht. Eine Lernrate, die in w_1 -Richtung vernünftig vorankommt, springt in w_2 -Richtung über die Schlucht hinaus; eine, die dort stabil ist, kriecht in w_1 -Richtung.

Der einfachste Ausweg ist, alle Features vorher auf Mittelwert 0 und Standardabweichung 1 zu bringen (*Standardisierung*, `sklearn.preprocessing.StandardScaler`, und beim Testset mit den Werten aus dem *Trainingsset* skalieren). Dass dieselbe Idee auch *innerhalb* eines Netzes hilft (Batch Normalization, Layer Normalization) und dass Optimierer wie Adam die Lernrate pro Parameter anpassen, sehen wir in späteren Kapiteln; der Grund ist immer dieser.

Wie so ein Loss-Gebirge für ein echtes neuronales Netz aussieht, zeigen z.B. Li, Xu, Taylor, Studer und Goldstein, *Visualizing the loss landscape of neural nets* (arXiv:1712.09913, 2017). Die Landschaften sind nicht konvex, es gibt viele lokale Minima, Sattelpunkte und Plateaus – und trotzdem funktioniert Gradientenabstieg erstaunlich gut.

Ausprobieren: Loss Landscape Explorer

losslandscape.com/explorer zeigt echte Loss-Landschaften trainierter Netze als begehbares Gelände, mit einem Gradientenabstiegsfad, den man selbst starten kann. Probieren Sie drei Lernraten: eine so kleine, dass der Pfad im ersten Hang stecken bleibt, eine mittlere, die ins Tal findet, und eine so große, dass der Pfad zwischen den Talwänden hin und her springt oder ganz herausfliegt. Was man daraus mitnimmt: Man kennt die Landschaft im echten Training nicht, man sieht nur den Loss-Wert; der Loss steigt auch mal an, ohne dass etwas kaputt ist; wenn er dauerhaft steigt oder **nan** wird, ist die Lernrate zu groß oder ein Neustart mit anderer Initialisierung fällig; und in der Praxis fängt man mit großer Lernrate an und macht sie später kleiner. Das Bild ist ein trianguliertes Gitter aus

wenigen Auswertungen und hängt an den Daten, mit anderen Trainingsdaten sähe dasselbe Netz anders aus.

3.3.4 Zurück zum Perzeptron

Die neue Perspektive erklärt auch die Perzeptronregel. Der „harte“ Perzeptron Loss $\sum_i \max(0, -y_i \text{sign}\langle w, x_i \rangle)$ hat ein Problem: Ableitung?!? Die Funktion ist stückweise konstant, der Gradient ist fast überall 0. Besser ist der „weiche“ Perzeptron Loss ohne das sign:

$$L_{\text{Perc}}(w) = \sum_{i=1}^N \max(0, -y_i \langle w, x_i \rangle). \quad (74)$$

Ein Summand ist 0, wenn x_i korrekt klassifiziert ist ($y_i \langle w, x_i \rangle > 0$), und sonst $-y_i \langle w, x_i \rangle > 0$, also proportional dazu, wie weit auf der falschen Seite x_i liegt. Die (schwache) Ableitung ist

$$L'_{\text{Perc}}(w) = \sum_{i \text{ falsch klassifiziert}} -y_i x_i, \quad (75)$$

die Summe über die falsch klassifizierten Samples. Die allgemeine Update-Regel $w_{\text{neu}} = w_{\text{alt}} - \rho L'(w)$ liefert für ein einzelnes (x, y_{true}) , also $N = 1$:

$$w_{\text{neu}} = w_{\text{alt}} \pm \rho x \quad \text{wenn } x \text{ falsch klassifiziert ist, sonst } w_{\text{neu}} = w_{\text{alt}}. \quad (76)$$

Das ist exakt die Perzeptronregel. Rosenblatt hat also, ohne es so zu nennen, stochastischen Gradientenabstieg auf dem Perzeptron Loss gemacht.

Und zurück zur logistischen Regression

Für die BCE lohnt sich dieselbe Rechnung, denn sie hat eine Pointe. Für ein einzelnes Sample (x, y) mit Logit $z = \langle w, x \rangle$ und $p = \sigma(z)$ ist

$$L(z) = -y \ln \sigma(z) - (1 - y) \ln(1 - \sigma(z)). \quad (77)$$

Mit $\sigma' = \sigma(1 - \sigma)$ und der Kettenregel:

$$\frac{dL}{dz} = -y \frac{\sigma(z)(1 - \sigma(z))}{\sigma(z)} + (1 - y) \frac{\sigma(z)(1 - \sigma(z))}{1 - \sigma(z)} = -y(1 - p) + (1 - y)p = p - y. \quad (78)$$

Alle Logarithmen und Sigmoiden heben sich gegenseitig weg; übrig bleibt der Vorhersagefehler. Mit $\frac{\partial z}{\partial w} = x$ ist der Gradient $(p - y)x$, und die Update-Regel lautet

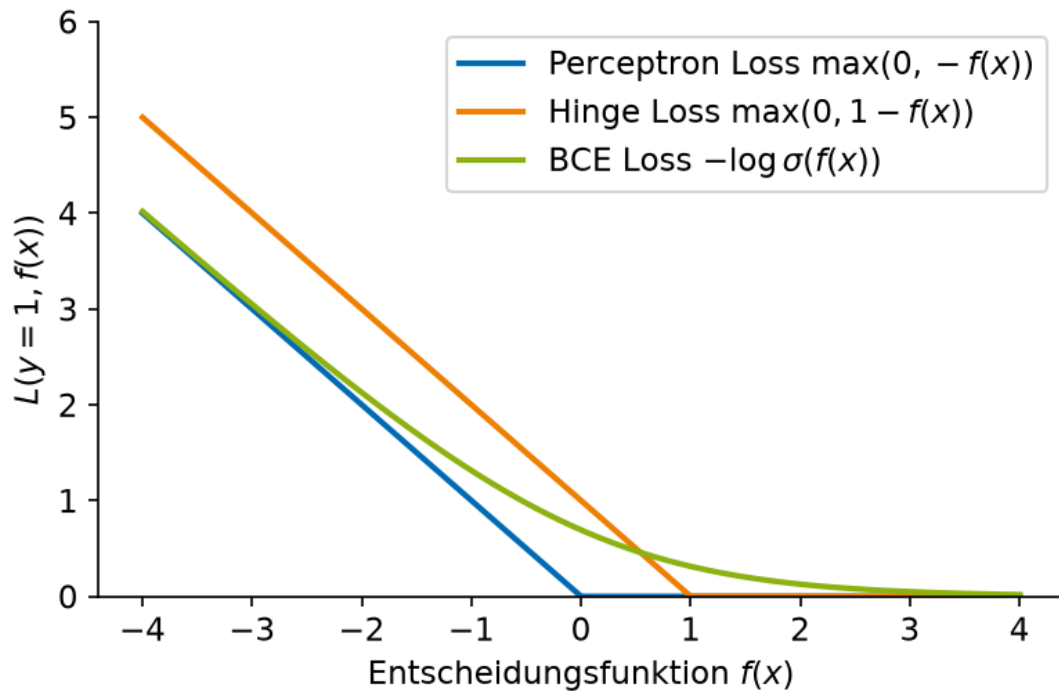
$$w_{\text{neu}} = w_{\text{alt}} - \rho (\sigma(\langle w_{\text{alt}}, x \rangle) - y) x = w_{\text{alt}} + \rho (y - f(x)) x. \quad (79)$$

Das ist buchstäblich die Perzeptronregel aus dem [Kapitel „Das Perzeptron trainieren“](#), nur mit $f(x) = \sigma(\langle w, x \rangle)$ statt $[\langle w, x \rangle > 0]$: Statt „falsch oder richtig“ zählt jetzt „um wie viel daneben“. Die Formel $p - y$ werden wir bei Backpropagation als Startwert der Rückwärtsrechnung wiedersehen; sie gilt genauso für Softmax mit Kreuzentropie.

Loss-Funktionen, die wir bisher gesehen haben

Für ein einzelnes Sample mit $y = 1$, aufgetragen gegen den Wert der Entscheidungsfunktion $f(x) = \langle w, x \rangle$:

- **Perzeptron Loss** $\max(0, -f(x))$: null, sobald $f(x) > 0$, davor linear.
- **Hinge Loss** $\max(0, 1 - f(x))$: null erst ab $f(x) \geq 1$, verlangt also einen *Sicherheitsabstand* (Margin) – das ist der Loss der Support Vector Machine.
- **BCE Loss** $-\log \sigma(f(x))$: glatt, wird nie exakt null, bestraft auch richtig klassifizierte Samples noch ein bisschen, wenn sie nahe an der Grenze liegen.



Drei Loss-Funktionen für ein Sample der Klasse $y = 1$ in Abhängigkeit vom Wert der Entscheidungsfunktion $f(x)$.

3.3.5 Take-Home Message

Ein einzelnes Neuron besteht aus einer Aktivierungsfunktion τ und einem Gewichtsvektor w (den Parametern). Inferenz:

$$y_{\text{pred}} = \tau(\langle w, x \rangle). \quad (80)$$

Training, mit Hyperparameter ρ , hängt von der Loss-Funktion $L(w)$ ab:

$$w_{\text{neu}} = w_{\text{alt}} - \rho L'(w_{\text{alt}}). \quad (81)$$

Für binäre Klassifikation nimmt man $\tau = \sigma$ und $L = L_{\text{BCE}}$. Das ist logistische Regression. In echt kennt man die Landschaft nicht und muss die Lernrate ausprobieren.

3.3.6 Nicht-stochastischer Gradientenabstieg??

Gradientenabstieg ist der Oberbegriff und gleichzeitig die spezifische Bezeichnung für: den ganzen Datensatz einlesen, dabei die Loss-Funktion auswerten und den Gradienten bestimmen, damit die Gewichte anpassen, und erneut den ganzen Datensatz einlesen usw. (auch *Batch Gradient Descent*).

Stochastischer Gradientenabstieg (SGD) ist das Anpassen der Gewichte mit einer *Schätzung* des Gradienten aus einem Teildatensatz – meist ist ein einzelnes Sample gemeint. Das ist zufällig gewählt (die Anordnung der Daten wird als zufällig angenommen), daher stochastisch. Der geschätzte Gradient ist verrauscht, aber im Mittel korrekt, und man kommt viel häufiger zu einem Update.

Was „im Mittel korrekt“ genau heißt: Der Loss $L(w) = \frac{1}{N} \sum_{i=1}^N \ell(x_i, y_i; w)$ ist ein Stichprobenmittel, also eine Schätzung des Erwartungswerts $\mathbb{E}[\ell(X, Y; w)]$ über die erzeugende Verteilung aus dem [Kapitel „Was ist Deep Learning?“](#). Das ist die Größe, die wir eigentlich klein haben wollen, denn sie misst den Fehler auf neuen Daten. Der Gradient eines Mini-Batches ist ebenfalls ein Stichprobenmittel; weil der Erwartungswert linear ist, ist sein Erwartungswert der Gradient des vollen Loss. Nach dem Gesetz der großen Zahlen ([DS-Skript](#)) liegt das Mittel über viele Batches nahe daran, auch wenn jeder einzelne Batch danebenliegt.

Wo steckt in diesem Verfahren überall Zufall? Im Startvektor, in den Daten, die wir als Stichprobe einer Verteilung modellieren, und in der Reihenfolge, in der die Samples verarbeitet werden. Deshalb

liefern zwei Trainingsläufe mit denselben Daten verschiedene Gewichte, und ein Schritt lässt sich nicht sinnvoll rückgängig machen; man kann nur neu starten.

Der Kompromiss ist **Mini-Batch SGD**: Eine kleine, handhabbare Menge von Samples (die *Batch Size*, z.B. 32 oder 64) wird gemeinsam verarbeitet und damit der Gradient approximiert. Das ist, was in der Praxis fast immer gemeint ist, wenn jemand „SGD“ sagt – auch `torch.optim.SGD` aus dem [Kapitel „Logistische Regression“](#) weiß nichts davon, wie viele Samples im Batch waren. Ein Durchlauf durch den gesamten Datensatz heißt **Epoche**.

3.3.7 Übung

Sei $x = (2, -1)$ fest und $L(w) := (1 - w \cdot x)^2$.

1. Bestimmen Sie $\frac{\partial L}{\partial w_1}$ und $\frac{\partial L}{\partial w_2}$ mit der Kettenregel. *Tipp: Äußere Funktion $u \mapsto u^2$, innere $w \mapsto 1 - w \cdot x$. Die Ableitung von $w \cdot x$ nach w_j ist x_j – mehr steckt nicht dahinter.*
2. Werten Sie den Gradienten bei $w = (0, 0)$ aus und führen Sie einen Schritt mit Lernrate $\rho = 0,1$ aus.
3. Berechnen Sie L vor und nach dem Schritt.

*Antwortprobe: L ist nach dem Schritt **exakt** 0 – bei $\rho = 0,1$ und $\|x\|^2 = 5$ landet man genau im Minimum. Kommt etwas anderes heraus, fehlt meist der Faktor 2 aus der Kettenregel.*

Weiterlesen

Prince, *UDL*, Kap. 6 (Fitting models: Gradientenabstieg, SGD, Lernrate); Deisenroth et al., *MML*, Kap. 5 (Vector calculus) und 7 (Continuous optimization). Beide Bücher sind frei verfügbar, siehe [Über dieses Skript](#).

3.4 Backpropagation

3.4.1 Ziel

Verstehen können, was im [TensorFlow Playground](#) geschieht.

Ausprobieren: TensorFlow Playground

Wählen Sie den XOR-Datensatz (die vier Quadranten), 2 Hidden Layers zu je 2 Neuronen, Aktivierung **tanh** oder **ReLU**, und setzen Sie die sechs Gewichte *von Hand* (auf eine Verbindung klicken, Wert eintippen), bis die vier Quadranten richtig gefärbt sind. Das geht ([Kapitel „Neuron und Perzeptron“](#) und [„Netze von Hand bauen“](#)), aber es ist mühsam. Dann Reset und Play. Zu beobachten: Der Gradientenabstieg findet meist *krumme* Gewichte, die trotzdem funktionieren; manche Neuronen sind redundant oder tragen nichts bei; die Lösung hängt von der zufälligen Initialisierung ab, und man kann während des Trainings einzelne Gewichte von Hand editieren und zusehen, wie das Netz sich erholt. Mit einem einzigen Hidden Layer aus vielen Neuronen sieht man, wie die Entscheidungsgrenze aus Stücken zusammengesetzt wird; das ist die Universalität aus dem [Kapitel „Universelle Approximation“](#).

Genau das Mühsame soll der Gradientenabstieg übernehmen. Dafür brauchen wir den Gradienten der Loss-Funktion nach *allen* Gewichten aller Schichten.

3.4.2 Training mehrerer Schichten: Motivation

Beispielsetup 1

Eingabedaten sind eindimensional, ein einzelnes Neuron mit Aktivierungsfunktion σ produziert eine Ausgabe, die stecken wir in ein zweites Neuron mit Aktivierungsfunktion id . Also:

$$y_{\text{pred}} = \langle w_{\text{zweit}}, \sigma(\langle w_{\text{erst}}, x \rangle) \rangle. \quad (82)$$



Beispielsetup 1: zwei Neuronen hintereinander.

Wir planen eine Regression mit $x, y \in \mathbb{R}$, daher nehmen wir den quadratischen Fehler – hier als reine Summe, ohne den Faktor $\frac{1}{N}$ des MSE aus dem [Kapitel „Gradientenabstieg“](#). Der Faktor verschiebt das Minimum nicht, nur die Größe des Gradienten und damit die passende Lernrate:

$$L(x, y; W) = \sum_{i=1}^N (y_i - y_{\text{pred},i})^2 \implies L'(W) = \sum_{i=1}^N 2(y_i - y_{\text{pred},i}) \left(-\frac{\partial}{\partial W} y_{\text{pred},i} \right). \quad (83)$$

Die Frage ist also nur: Was ist $\frac{\partial}{\partial W} y_{\text{pred}}$, die Ableitung der Netzausgabe nach den Gewichten?

Vorbereitung

Nehmen wir $W = (w_{\text{erst}}, w_{\text{zweit}})$, so können wir y_{pred} als Funktion von W auffassen, $y_{\text{pred}}(W) = \langle w_{\text{zweit}}, \sigma(\langle w_{\text{erst}}, x \rangle) \rangle$, und die Ableitung nach W wie zuvor in Komponenten berechnen. Dafür brauchen wir zwei Bausteine:

- Was ist die Ableitung von $\sigma(f(w))$ nach w ? Kettenregel: $\frac{d}{dw} \sigma(f(w)) = \sigma'(f(w)) f'(w)$.

- Was ist die Ableitung von $\langle w, x \rangle$ nach w , und von $\langle v, f(w) \rangle$ nach w ? Es ist $\frac{d}{dw} \langle w, x \rangle = x$ und $\frac{d}{dw} \langle v, f(w) \rangle = \langle v, f'(w) \rangle$.

Damit:

$$\frac{\partial}{\partial w_{\text{erst}}} y_{\text{pred}}(W) = \langle w_{\text{zweit}}, \sigma'(\langle w_{\text{erst}}, x \rangle) x \rangle, \quad \frac{\partial}{\partial w_{\text{zweit}}} y_{\text{pred}}(W) = \sigma(\langle w_{\text{erst}}, x \rangle). \quad (84)$$

Die Update-Regeln

Eingesetzt in L' und in die Gradientenabstiegs-Regel $W_{\text{neu}} = W_{\text{alt}} - \rho L'(W_{\text{alt}})$:

$$w_{\text{zweit,neu}} = w_{\text{zweit,alt}} + \rho \sum_{i=1}^N 2 \left(y_i - \langle w_{\text{zweit,alt}}, \sigma(\langle w_{\text{erst,alt}}, x_i \rangle) \rangle \right) \sigma(\langle w_{\text{erst,alt}}, x_i \rangle), \quad (85)$$

$$w_{\text{erst,neu}} = w_{\text{erst,alt}} + \rho \sum_{i=1}^N 2 \left(y_i - \langle w_{\text{zweit,alt}}, \sigma(\langle w_{\text{erst,alt}}, x_i \rangle) \rangle \right) \langle w_{\text{zweit,alt}}, \sigma'(\langle w_{\text{erst,alt}}, x_i \rangle) x_i \rangle. \quad (86)$$

Das funktioniert, aber es ist unübersichtlich – und man sieht schon, dass Teilausdrücke mehrfach vorkommen: der Residuenterm $y_i - y_{\text{pred},i}$ und die Aktivierung $\sigma(\langle w_{\text{erst}}, x_i \rangle)$ der ersten Schicht tauchen in beiden Regeln auf.

Beispielsetups 2 und 3: mehr Dimensionen, mehr Neuronen

Setup 2: Eingabedaten sind zweidimensional, sonst wie oben. Jetzt neu: $w_{\text{erst}} \in \mathbb{R}^2$. An den Formeln ändert sich nichts, die Skalarprodukte sind nur länger.

Setup 3: Eingabedaten sind zweidimensional, *zwei* Neuronen mit Aktivierungsfunktion σ und Gewichten $u, v \in \mathbb{R}^2$ produzieren Ausgaben, die stecken wir in ein drittes Neuron mit Aktivierungsfunktion id und Gewichten $w \in \mathbb{R}^2$:

$$y_{\text{pred}} = \langle w, (\sigma \langle u, x \rangle, \sigma \langle v, x \rangle) \rangle. \quad (87)$$

Layer

Mehrere Neuronen, die parallel verarbeitet werden, bilden eine **Schicht** (Layer). Mehrmals lineare Regression ist *multiple linear Regression* ist ein **Linear Layer** ist eine Matrix, deren Zeilen die Gewichtsvektoren sind. Schreiben wir also u, v als Zeilen in eine Matrix $A \in \mathbb{R}^{2 \times 2}$ und benutzen σ auf Vektoren komponentenweise, so ist

$$y_{\text{pred}} = \langle w, \sigma(Ax) \rangle = w^T \sigma(Ax). \quad (88)$$

Die Ableitung nach einem einzelnen Gewicht $u_1 = A_{11}$ ist $\frac{\partial}{\partial u_1} = \frac{\partial}{\partial A_{11}}$, und für die ganze Matrix:

$$\frac{\partial}{\partial A} y_{\text{pred}} = \frac{\partial}{\partial A} w^T \sigma(Ax) = w^T \frac{\partial}{\partial A} \sigma(Ax) = w^T \sigma'(Ax) x, \quad \frac{\partial}{\partial w} y_{\text{pred}} = \frac{\partial}{\partial w} w^T \sigma(Ax) = \sigma(Ax). \quad (89)$$

(Wobei bei $w^T \sigma'(Ax) x$ die Dimensionen passend zu lesen sind: $\sigma'(Ax)$ ist eine Diagonalmatrix, und das Ergebnis ist eine 2×2 -Matrix, deren Eintrag (j, k) gleich $w_j \sigma'((Ax)_j) x_k$ ist.)

Idee: Berechne für y_{pred} gleichzeitig die Daten für den Gradienten. Nach der Kettenregel $\frac{d}{dx} f(g(x)) = f'(g(x)) g'(x)$ mit $g(x) = \sigma(Ax)$ und $f(z) = w^T z$ berechnen wir zuerst g und g' vom ersten Layer, dann damit den nächsten Layer, usw. bis zum letzten. Die Zwischenergebnisse der Vorwärtsrechnung sind genau das, was die Rückwärtsrechnung braucht.

3.4.3 Backpropagation konkret

Beispiel 4: ein Netz mit drei Schichten

Wir betrachten

$$f(x) = b_3 + W_3 \tanh(b_2 + W_2 \tanh(b_1 + W_1 x)) \quad (90)$$

und schreiben die Vorwärtsrechnung (*forward computation*) schichtweise auf:

$$\begin{aligned}
z_1 &= x && \text{(input layer)} \\
z_2 &= \tanh(b_1 + W_1 z_1) && \text{(2nd layer)} \\
z_3 &= \tanh(b_2 + W_2 z_2) && \text{(3rd layer)} \\
y_{\text{pred}} &= b_3 + W_3 z_3 && \text{(output layer)} \\
r &= y_{\text{pred}} - y && \text{(residual)} \\
E &= \frac{1}{2} r^T r && \text{(squared error)}
\end{aligned} \tag{91}$$

Der Faktor $\frac{1}{2}$ ist reine Bequemlichkeit, er kürzt sich gleich gegen die 2 aus der Ableitung des Quadrats.

Rückwärts differenzieren

Wir verwenden die Notation der Differentiale: dE ist die lineare Änderung von E bei einer infinitesimalen Änderung der Parameter, und $\nabla_b E$ bezeichnet den Gradienten von E nach b (mit der Konvention $dE = \langle \nabla_b E, db \rangle$). Wir arbeiten uns von hinten nach vorn (*backwards differentiation*).

Output layer. Es ist

$$dE = d\left(\frac{1}{2} r^T r\right) = \frac{1}{2} d(r^T r) = r^T dr = r^T dy_{\text{pred}}, \tag{92}$$

denn y ist konstant. Weiter ist $dy_{\text{pred}} = d(b_3 + W_3 z_3) = db_3 + dW_3 z_3 + W_3 dz_3$. Für die Ableitung nach b_3 betrachten wir nur den ersten Term: $dE = r^T db_3$, also

$$\nabla_{b_3} E = r. \tag{93}$$

Für die Ableitung nach W_3 ist das Ergebnis

$$\nabla_{W_3} E = r z_3^T. \tag{94}$$

Vertiefung: der Spur-Trick

Woher kommt das? Aus $dE = r^T dW_3 z_3$ soll ein Ausdruck der Form $\langle \nabla_{W_3} E, dW_3 \rangle = \text{tr}((\nabla_{W_3} E)^T dW_3)$ werden. Der Skalar $r^T dW_3 z_3$ ist seine eigene Spur, und die Spur darf zyklisch vertauscht werden: $\text{tr}(r^T dW_3 z_3) = \text{tr}(z_3 r^T dW_3)$. Der Vergleich mit $\text{tr}((\nabla_{W_3} E)^T dW_3)$ liefert $(\nabla_{W_3} E)^T = z_3 r^T$. Wer das nicht mag, rechnet komponentenweise: $\partial E / \partial (W_3)_{jk} = r_j (z_3)_k$, dasselbe Ergebnis.

Das ist ein *äußeres Produkt*: Der Gradient nach der Gewichtsmatrix ist Residuum mal Eingabe der Schicht, transponiert – so wie beim einzelnen Neuron der Gradient nach w gleich Fehler mal x war.

Nächste Schicht. Jetzt können wir b_3 und W_3 als Konstanten sehen: $dE = r^T W_3 dz_3$ + irrelevante Terme. Mit der Kettenregel gilt $\frac{d}{dz} \tanh(f(z)) = (1 - \tanh(f(z))^2) f'(z)$, also

$$dz_3 = d \tanh(b_2 + W_2 z_2) = (1 - z_3^2) \odot (db_2 + dW_2 z_2 + W_2 dz_2), \tag{95}$$

wobei z_3^2 und $\odot$ komponentenweise zu lesen sind. Für b_2 : $dE = r^T W_3 \text{diag}(1 - z_3^2) db_2$, also

$$\nabla_{b_2} E = (1 - z_3^2) \odot W_3^T r, \quad \nabla_{W_2} E = (\nabla_{b_2} E) z_2^T. \tag{96}$$

Und noch eine Schicht. Nach demselben Muster:

$$\nabla_{b_1} E = (1 - z_2^2) \odot W_2^T (\nabla_{b_2} E), \quad \nabla_{W_1} E = (\nabla_{b_1} E) z_1^T. \tag{97}$$

Das Muster

Zusammengefasst, mit $\delta_k := \nabla_{b_k} E$ als „Fehlersignal“ der Schicht k :

$$\delta_3 = r, \quad \delta_2 = (1 - z_3^2) \odot W_3^T \delta_3, \quad \delta_1 = (1 - z_2^2) \odot W_2^T \delta_2, \quad \nabla_{W_k} E = \delta_k z_k^T. \tag{98}$$

Das Fehlersignal wird von Schicht zu Schicht *rückwärts propagiert*: mit der transponierten Gewichtsmatrix multipliziert und mit der Ableitung der Aktivierungsfunktion gewichtet. Man nutzt die Zwischenschritte z_k aus der Vorwärtsrechnung, um den vollen Gradienten effizient zu berechnen – ein Vorwärtsdurchlauf, ein Rückwärtsdurchlauf, fertig. Der Aufwand ist etwa das Doppelte einer Inferenz, unabhängig von der Anzahl der Parameter. Das ist **Backpropagation**, und danach kommt Gradientenabstieg.

Genau das tut `loss.backward()` in PyTorch ([Kapitel „Autograd und Berechnungsgraph“](#)): Beim Vorwärtsrechnen merkt sich jeder Tensor mit `requires_grad`, aus welcher Operation er entstanden ist (der *Berechnungsgraph*), und `backward` läuft diesen Graphen rückwärts ab und schreibt die Gradienten in die `.grad`-Attribute der Parameter. `optimizer.step()` macht dann $W \leftarrow W - \rho \nabla_W E$.

Matrix-Differentialkalkül: Literaturtipps

Achtung: Das ist nicht notwendig für diese Veranstaltung, das ist weiterführend.

- Thomas P. Minka, *Old and new matrix algebra useful for statistics*, Notes, 2000.
- *The Matrix Cookbook* (Petersen & Pedersen).
- *Matrix derivatives cheat sheet*.
- Wikipedia: *Loss functions for classification*.
- Michael Nielsens Buchkapitel zu Backpropagation (sehr mathematisch): neuralnetworksanddeeplearning.com/chap2.html.

3.4.4 Backpropagation als Induktion

Noch einmal systematisch, als Wiederholung und Zusammenfassung aus der Übung – denn Backpropagation ist Klausurstoff, und die Formulierung als Induktion ist die, mit der man jede Architektur erschlägt.

Warum Backpropagation funktioniert

1. Im **Forward Pass** speichern wir für jede Rechenoperation das Zwischenergebnis und merken uns die Ableitung der einzelnen Rechenoperation und auf welche vorangehenden Ergebnisse wir zurückgreifen müssen (Konstruktion des DAG).
2. Im **Backward Pass** können wir für jede Rechenoperation jeweils nach den Gewichten ableiten, die für diese Operation relevant sind. Wenn wir nach Gewichten ableiten, von denen die vorangegangenen Rechenoperationen abhängen, müssen wir die Kettenregel anwenden. Dazu benötigen wir nur die Ableitung der neuen Rechenoperation (gespeichert im Forward Pass), den Wert der Rechenoperation (gespeichert im Forward Pass) und die Ableitung der vorangegangenen Rechenoperationen.

Mit vollständiger Induktion sehen wir also: Wir können alle Ableitungen berechnen mit *einem* Durchlauf durch den DAG. Insbesondere können wir die Loss-Funktion (das „Ende“ des DAG) nach allen Gewichten ableiten und so den vollständigen Gradienten berechnen.

Der Induktionsschritt, systematisch

Wir bezeichnen den Input als $x \in \mathcal{X}$. Die „bisherigen Rechenoperationen“ sind eine Abbildung $Y: \mathcal{X} \rightarrow \mathcal{Y}$, also $y := Y(x; w_Y)$, deren Ableitungen $\frac{dY}{dw_Y}(x; w_Y)$ wir per Induktion kennen. Die „nächste Rechenoperation“ ist eine Abbildung $Z: \mathcal{Y} \rightarrow \mathcal{Z}$, deren Ableitungen $\frac{dZ}{dy}$ und $\frac{dZ}{dw_Z}$ wir kennen. Die Kettenregel erlaubt uns, $Z(Y(x; w_Y); w_Z)$ nach w_Y abzuleiten:

$$\frac{d}{dw_Y} Z(Y(x; w_Y); w_Z) = \frac{dZ}{dy}(y; w_Z) \cdot \frac{dY}{dw_Y}(x; w_Y), \quad \frac{d}{dw_Z} Z(Y(x; w_Y); w_Z) = \frac{dZ}{dw_Z}(y; w_Z). \quad (99)$$

Spezialfälle: Wenn die nächste Rechenoperation der Loss ist, gibt es keine „neuen Gewichte“, dann ist nur die Ableitung nach „bisherigen Gewichten“ relevant. Der Induktionsanfang ist gegeben, indem Y die Identitätsfunktion ist (Ableitung ist 1). In knapperer Notation, mit Kleinbuchstaben als Funktionsnamen:

$$\frac{dz}{dw} = \frac{dz}{dy} \cdot \frac{dy}{dw}. \quad (100)$$

Nicht konkret genug? Ein weiteres einfaches Beispiel

Wir wählen als Input $x \in \mathcal{X} \subseteq \mathbb{R}^n$. Im Beispiel soll $y := y(x; w) := w \cdot x$ sein, also ein linearer Layer. Die „nächste Rechenoperation“ ist $z: \mathbb{R} \rightarrow \mathbb{R}$, und zwar $z := z(y) := \text{ReLU}(y) = \max(0, y)$. Die Ableitung nach y ist $[y > 0]$. Aus $\frac{dz}{dw} = \frac{dz}{dy} \cdot \frac{dy}{dw}$ wird also

$$\frac{dz}{dw} = [y > 0] \cdot x. \quad (101)$$

3.4.5 Übung

Ein Netz $x \rightarrow y = \text{ReLU}(v x) \rightarrow z = \text{ReLU}(w y)$ mit $v = 2$, $w = -1$, Eingabe $x = 3$ und Ziel $t = 1$; Loss $L = \frac{1}{2}(z - t)^2$.

1. Forward Pass: Berechnen Sie y , z und L .
2. Berechnen Sie $\frac{\partial L}{\partial w}$ und $\frac{\partial L}{\partial v}$. *Tipp: $\frac{\partial L}{\partial z} = z - t$, die schwache Ableitung von ReLU an der Stelle a ist $[a > 0]$, und der Gradient eines Gewichts ist Eingabe des Neurons mal Ableitung des Fehlers nach seiner Ausgabe.*
3. Setzen Sie $w = +1$ und rechnen Sie noch einmal. Was hat sich an $\frac{\partial L}{\partial v}$ qualitativ geändert, und warum?

Antwortprobe: In genau einem der beiden Fälle ist $\frac{\partial L}{\partial v} = 0$.

Und eine größere, wenn Sie Lust haben: Nehmen Sie das Beispiel aus dem Abschnitt „Nicht konkret genug?“ ($y = \langle a, x \rangle$, dann $z = \text{ReLU}(y)$) und geben Sie der ersten Schicht ein zweites Neuron, also eine zweite Zeile $a' \in \mathbb{R}^n$. Der Loss ist dann $\|z - z_{\text{ground truth}}\|^2$ mit $z \in \mathbb{R}^2$. Leiten Sie nach beiden Zeilen ab und prüfen Sie, dass die beiden Gradienten sich nicht gegenseitig beeinflussen.

Weiterlesen

Prince, *UDL*, Kap. 7 (Gradients and initialization), dort Backpropagation mit derselben Rückwärts-Notation. Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).

3.5 Autograd und Berechnungsgraph

Im [Kapitel „Backpropagation“](#) haben wir Gradienten von Hand rückwärts durch ein Netz gerechnet. PyTorch tut genau dasselbe – automatisch, für beliebige Architekturen, und mit demselben Muster: Zwischenergebnisse im Vorwärtsthroughlauf speichern, Kettenregel im Rückwärtsthroughlauf. Dieses Kapitel schaut hinter die Kulissen von `loss.backward()`.

3.5.1 Ein Beispiel im Kleinen

Ein Blick darauf, wie PyTorch Backpropagation *tatsächlich* macht. Wir bauen ein sehr kleines „Netzwerk“:

```
import torch

x = torch.tensor(3.0)                # Input (Konstante)
w = torch.tensor(2.0, requires_grad=True) # Gewicht: "Beobachte mich!"
b = torch.tensor(1.0, requires_grad=True) # Bias
z = w * x                            # z = 6
y = z + b                            # y = 7
L = y ** 2                            # Loss = 49
```

Der **Berechnungsgraph** (ein DAG, directed acyclic graph) entsteht erst beim Ausführen: Jeder Tensor, der aus einer Operation mit einem `requires_grad=True`-Tensor entstanden ist, trägt eine `grad_fn`, und man kann den Graphen von hinten nach vorn ablaufen:

```
print(L.grad_fn)                    # <PowBackward0>
print(L.grad_fn.next_functions[0][0]) # <AddBackward0>
print(L.grad_fn.next_functions[0][0].next_functions)
# ((<MulBackward0>, 0), (<AccumulateGrad>, 0)) -- z, und das Blatt b
```

Die `grad_fn` weiß, wie man die Operation ableitet, und hält Referenzen auf die *saved tensors*, die gespeicherten Zwischenergebnisse – genau die z_k aus dem [Kapitel „Backpropagation“](#). `AccumulateGrad` sind die Blätter des Graphen, die Parameter w und b ; die topologische Sortierung ist implizit.

`L.backward()` ist der rekursive Aufruf der Kettenregel, initial mit $\frac{dL}{dL} = 1$:

$$\frac{dL}{dw} = \frac{dL}{dy} \cdot \frac{dy}{dz} \cdot \frac{dz}{dw} = 2y \cdot 1 \cdot x = 14 \cdot 3 = 42, \quad \frac{dL}{db} = 2y \cdot 1 = 14. \quad (102)$$

```
L.backward()
print(w.grad, b.grad) # tensor(42.), tensor(14.)
```

Danach steht das Ergebnis in `w.grad` und `b.grad`. Ein Schritt Gradientenabstieg von Hand, das ist alles, was `optimizer.step()` tut:

```
lr = 0.01
with torch.no_grad():
    w -= lr * w.grad # 2.0 - 0.01 * 42 = 1.58
    b -= lr * b.grad # 1.0 - 0.01 * 14 = 0.86
    w.grad.zero_()   # sonst wird beim nächsten backward() akkumuliert
    b.grad.zero_()
```

Der Optimizer-Schritt muss unter `torch.no_grad()` passieren, damit der Update selbst nicht Teil des Graphen wird, und die Gradienten müssen danach mit `zero_()` zurückgesetzt werden – sonst *akkumuliert* PyTorch sie beim nächsten `backward()`. Das ist der Grund für `optimizer.zero_grad()` in jeder Trainingsschleife.

3.5.2 Freezing

Setzt man `w.requires_grad = False`, so hat `z = w * x` keine `grad_fn` mehr, der Graph ist an dieser Stelle abgeschnitten:

```
w.requires_grad = False
z = w * x
print(z.grad_fn)        # None -- Blatt abgeschnitten
print((z + b).grad_fn)  # <AddBackward0> -- b wird weiter beobachtet
```

Ein Optimizer verändert w nicht mehr, b aber schon. So friert man beim Transfer Learning Schichten eines vortrainierten Modells ein – und spart dabei Speicher, weil für eingefrorene Parameter keine Gradienten gehalten werden.

3.5.3 Geteilte Gewichte: Backprop durch eine Faltung

Wie funktioniert Backprop, wenn Gewichte geteilt werden wie bei einem Faltungslayer? Der Kernel [0.5] wird über den Input [1, 2, 3] geschoben, $o_i = x_i \cdot k$, und der Loss ist $o_1 + o_2 + o_3$:

```
x = torch.tensor([1.0, 2.0, 3.0])
k = torch.tensor([0.5], requires_grad=True)
o1, o2, o3 = x[0] * k, x[1] * k, x[2] * k  # von Hand, wie nn.Conv1d
loss = o1 + o2 + o3
loss.backward()
print(k.grad)  # tensor([6.])
```

Der Gradient nach k ist die *Summe* der Beiträge aller Positionen, $\frac{dL}{dk} = x_1 + x_2 + x_3 = 6$: Ein geteilter Parameter bekommt die Gradienten aller seiner Verwendungen aufsummiert – im Graphen führen mehrere Kanten zu demselben Blatt.

Example 3.3 (Backpropagation durch Faltung und ReLU von Hand). *Ein Neuron führt eine 1D-Faltung gefolgt von ReLU aus: Eingangssignal $x = [2, 0, -1, 4]$, Kernel $k = [k_0, k_1] = [1, -1]$, Bias $b = 0$, „valid“ ohne Padding, Stride 1. Die Pre-Aktivierung an Position i ist $z_i = x_i k_0 + x_{i+1} k_1 + b$, die Ausgabe $a_i = \text{ReLU}(z_i)$.*

Forward: $z = [2 \cdot 1 + 0 \cdot (-1), 0 + (-1)(-1), -1 + 4 \cdot (-1)] = [2, 1, -5]$, also $a = [2, 1, 0]$.

Loss: $L = \frac{1}{2} \sum_i (a_i - t_i)^2$ mit Target $t = [1, 2, 0]$, also $\frac{\partial L}{\partial a} = a - t = [1, -1, 0]$.

Backward nach k_0 : Die ReLU-Maske ist $[z_i > 0] = [1, 1, 0]$; nur die Positionen 0 und 1 leiten den Gradienten weiter. Da k_0 an jeder Position mit x_i multipliziert wird, ist $\frac{\partial z_i}{\partial k_0} = x_i$, und die Beiträge summieren sich:

$$\frac{\partial L}{\partial k_0} = \sum_i (a_i - t_i) [z_i > 0] x_i = 1 \cdot 1 \cdot 2 + (-1) \cdot 1 \cdot 0 + 0 = 2. \quad (103)$$

Update mit Lernrate $\rho = 0,1$: $k_0 \leftarrow 1 - 0,1 \cdot 2 = 0,8$. Analog $\frac{\partial L}{\partial k_1} = 1 \cdot 0 + (-1)(-1) + 0 = 1$.

3.5.4 Übung

1. Ein Modell hat drei Layer. Vor dem Training wird für Layer 1 und 2 `requires_grad = False` gesetzt. Welche Tensoren bekommen im Backward Pass einen `.grad`, welche nicht – und wo wird trotzdem noch gerechnet? *Tipp: Zeichnen Sie den Graphen und verfolgen Sie die Pfeile rückwärts, von der Loss-Funktion zu den Blättern. Fragen Sie sich bei jedem Layer: Muss der Gradient hier überhaupt noch hindurch, damit weiter vorne etwas ankommt?*
2. Ein Gewicht k wird an drei Stellen im Graphen benutzt (geteilte Gewichte, wie bei einer Faltung). Wie viele Beiträge hat $\frac{\partial L}{\partial k}$, und wie werden sie verrechnet?

Antwortprobe: Bei (1) spart das Einfrieren Speicher, aber nicht jede Rechnung.

3.6 Aktivierungsfunktionen

3.6.1 Aktivierungsfunktionen und weitere Layer

Ein Neuron „forward“ ist $\tau(\langle w, x \rangle)$. Beim Perzeptron ist $\tau = \text{sign}$, bei der logistischen Regression $\tau = \sigma$. Allgemein ist τ beliebig – aber ableitbar, sonst funktioniert Backpropagation nicht. *Überall* differenzierbar muss es dafür nicht sein: ReLU hat bei 0 einen Knick und ist trotzdem die meistbenutzte Aktivierungsfunktion überhaupt. Wie das zusammengeht, klärt der nächste Abschnitt.

Neues Ziel: Funktionen lernen

Bisher war das Ziel Klassifikation. Neues Ziel: Das neuronale Netz soll eine Funktion lernen (Regression), und unser Modell von der Funktion ist *stückweise linear*. Welches τ ist dafür geeignet?

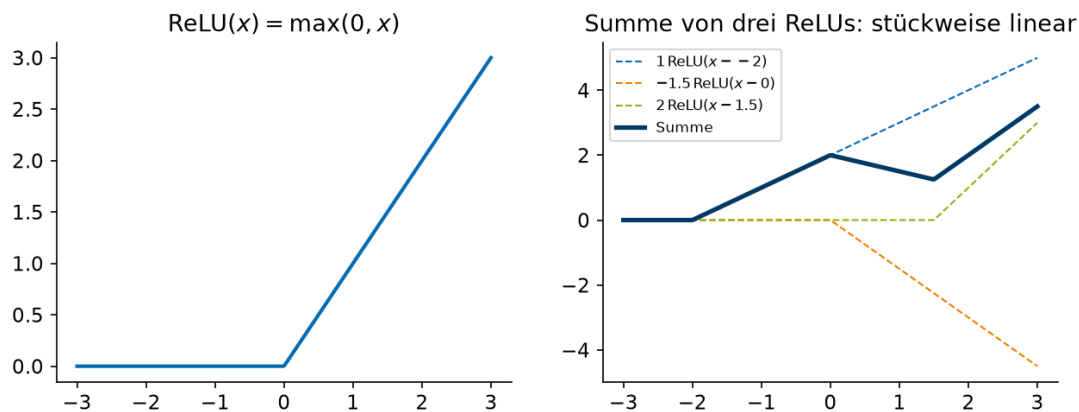
Definition 3.5 (ReLU). *Die Rectified Linear Unit ist*

$$\text{ReLU}(y) := \max(0, y). \quad (104)$$

Ein Neuron mit $\tau = \text{ReLU}$ heißt *ReLU-Neuron*.

Ableitung von $\max(0, y)$ nach y : Für $y > 0$ ist $\max(0, y) = y$, also ist die Ableitung 1. Für $y < 0$ ist $\max(0, y) = 0$, also ist die Ableitung 0. Für $y = 0$ ist die Ableitung nicht definiert – wir *definieren* sie als 0, das spart Rechnungen. (Dass das erlaubt ist, steckt im Begriff der *schwachen Ableitung*: Sie darf an einzelnen Punkten beliebig sein.)

Wieso ist ReLU geeignet? Lineare Abbildung + ReLU ist die Projektion auf einen Halbraum, denn ReLU ist genau die Projektion von $\mathbb{R}$ auf $[0, \infty)$, und ein Halbraum ist eine Seite einer Hyperebene. Ein ReLU-Neuron $\max(0, \langle w, x \rangle + b)$ ist also null auf der einen Seite der Hyperebene $\langle w, x \rangle + b = 0$ und linear ansteigend auf der anderen. Summiert man mehrere solche „Knickfunktionen“ mit Gewichten, bekommt man stückweise lineare Funktionen, deren Knickstellen auf den Hyperebenen liegen – und mit genug Neuronen kann man damit jede stetige Funktion beliebig gut approximieren. Beispiel an der Tafel; im [Kapitel „Netze von Hand bauen“](#) bauen wir so x^2 .



Links die ReLU, rechts eine gewichtete Summe von drei verschobenen ReLUs: eine stückweise lineare Funktion mit Knicken an den Verschiebungen.

Layer, nochmal

Mehrere Neuronen, die parallel verarbeitet werden, bilden eine Schicht. Ein **Linear Layer** ist eine Matrix (Zeilen = Gewichtsvektoren) plus Bias. Ein **ReLU Layer** ist eine Schicht, in der jedes Neuron die Aktivierungsfunktion ReLU nutzt, also Linear Layer + komponentenweises ReLU. In PyTorch: `nn.Sequential(nn.Linear(n, m), nn.ReLU())`.

Problem bei ReLU: sterbende Neuronen

Wenn ein Neuron immer 0 liefert, weil die Eingabe immer negativ ist, „fließt“ keine Information mehr durch das Neuron während der Backpropagation – die Ableitung ist 0, also ist δ für dieses Neuron 0, also ändern sich seine Gewichte nie wieder. Das Neuron ist tot. Zwei Lösungen:

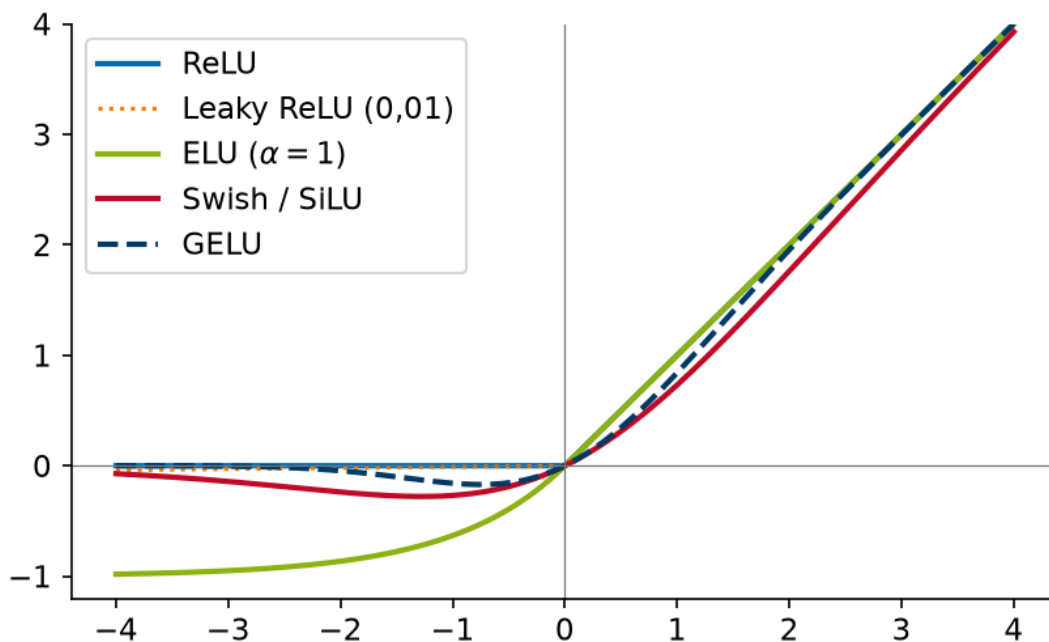
Lösung 1: Statt $\tau = \max(0, y)$ verwende $\tau(y) = f(y)[y < 0] + g(y)[y \geq 0]$ mit passenden f, g , sodass auch für $y < 0$ die Ableitung nicht null ist. Mit $f(y) := \alpha(e^y - 1)$ und $g(y) := y$ heißt τ **ELU** (Exponential Linear Unit); mit $f(y) := 0,01 y$ heißt es Leaky ReLU.

Lösung 2: Statt $\max(0, y)$ verwende ein glattes τ , sodass für $y \rightarrow -\infty$ gilt $\tau(y) \rightarrow 0$ und für $y \rightarrow +\infty$ gilt $\tau(y) \rightarrow +\infty$, aber ohne harten Knick:

$$\tau(y) := y \sigma(y) \quad (\text{Swish, auch SiLU}), \quad \tau(y) := y \Phi(y) \quad (\text{GELU, Gaussian Error Linear Unit}), \quad (105)$$

wobei Φ die Verteilungsfunktion der Standardnormalverteilung ist. Beide sind im negativen Bereich nicht strikt null, sondern haben eine kleine Delle unter der Achse, und die Ableitung verschwindet nur asymptotisch. GELU steht in den Transformatoren der BERT- und GPT-2-Generation und in Vision Transformatoren; die heutigen großen Sprachmodelle (Llama, Mistral, Qwen) nehmen stattdessen Swish/SiLU, eingebaut in einen *gated* Feed-Forward-Block, den man SwiGLU nennt.

Welche Aktivierungsfunktionen in der Praxis genutzt wurden, listete [paperswithcode](#) – die Seite gibt es nicht mehr, der Link führt auf den Schnappschuss vom Juli 2025 im Internet Archive.



Aktivierungsfunktionen im Vergleich. ELU, Swish und GELU sind im negativen Bereich nicht konstant null.

3.6.2 Übung

Ein Neuron $x \mapsto \text{ReLU}(w \cdot x + b)$ mit $w = (-1, -1)$ und $b = -3$ sieht nur Trainingsdaten mit nichtnegativen Einträgen.

1. Zeigen Sie: Die Ausgabe ist für **alle** diese Daten 0.
2. Welchen Wert hat der Gradient nach w und b dann? Was folgt daraus für das Training?

Antwortprobe: Bei (2) ist die Antwort für w und b dieselbe.

Weiterlesen

Prince, *UDL*: Die Aktivierungsfunktionen stehen in Kap. 3 (Abbildung 3.13 und der Abschnitt *Notes* am Kapitelende) und in Abschnitt 20.2.5 (Activation functions); zu Initialisierung und sterbenden Neuronen Kap. 7 (Gradients and initialization). Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).

3.7 Netze von Hand bauen

3.7.1 Das Geheimnis um Deep Learning lüften

Warum – und wie genau – funktioniert Deep Learning so gut? Die Zutaten, die wir bisher haben:

- Das einzelne Neuron = Skalarprodukt + Aktivierungsfunktion.
- Gradientenabstieg: Lernen in Richtung der Ableitung einer Loss-Funktion.
- Backpropagation: clever die Ableitung berechnen für tiefe Netze.

Was noch fehlt, und was wir in diesem und den nächsten Kapiteln angehen:

- Sinnvolle Loss-Funktionen festlegen: Regression mit MSE, (binäre) Klassifikation mit (B)CE; BCE vs. NLL vs. D_{KL} .
- Architekturen neben einfachen MLPs, wie z.B. Autoencoder – und damit auch Representation Learning und Transfer Learning ([Kapitel „Autoencoder“](#) und folgende).
- Regularisierung und Tricks gegen verschwindende Gradienten ([Kapitel „Aktivierungsfunktionen“](#), Dropout und Batch-Normalisierung im Autoencoder-Kapitel).
- Hyperparameter wählen und optimieren; alles hängt von den Daten ab ...
- Und ganz grundsätzlich: Elementare Funktionen sind leicht approximierbar – das jetzt, zuerst von Hand (dieses Kapitel), dann als Satz ([Kapitel „Universelle Approximation“](#)).

3.7.2 Parameter von Hand wählen

Bevor wir Netze trainieren, überzeugen wir uns davon, dass die gesuchten Gewichte überhaupt *existieren*: Wir bauen kleine Netze von Hand.

Easy mode: Boolesche Variablen

Seien $x, y \in \{0, 1\} = \{\text{FALSE}, \text{TRUE}\}$ und $f: \mathbb{R}^2 \rightarrow \mathbb{R}$, $f(x, y) := x \text{ OR } y$, also $f(0, 0) = 0$, $f(1, 0) = 1$, $f(0, 1) = 1$, $f(1, 1) = 1$. Welches neuronale Netz berechnet als Inferenz die Funktion f ?

Idee: Schreibe f als Verkettung von gewichteten Summen (= Skalarprodukten) und Aktivierungsfunktionen (z.B. $[x > 0]$ oder $\text{ReLU}(x) := \max(0, x)$). Dabei müssen wir OR *interpolieren* – entscheiden, was z.B. $f(0,5, 0)$ sein soll – und *extrapolieren* – entscheiden, was z.B. $f(-1, 2)$ sein soll. Denn ein neuronales Netz ist auf ganz $\mathbb{R}^2$ definiert, nicht nur auf den vier Ecken.

Easiest mode: eindimensional

Sei $x \in \mathbb{R}$ mit der Interpretation $(x \leq 0) = \text{FALSE}$, $(x \geq 1) = \text{TRUE}$, und $n: \mathbb{R} \rightarrow \mathbb{R}$, $n(x) := \text{NOT } x$. Also $n(0) = 1$, $n(1) = 0$, allgemeiner $n(x) \leq 0$ falls $x \geq 1$. Die Funktion

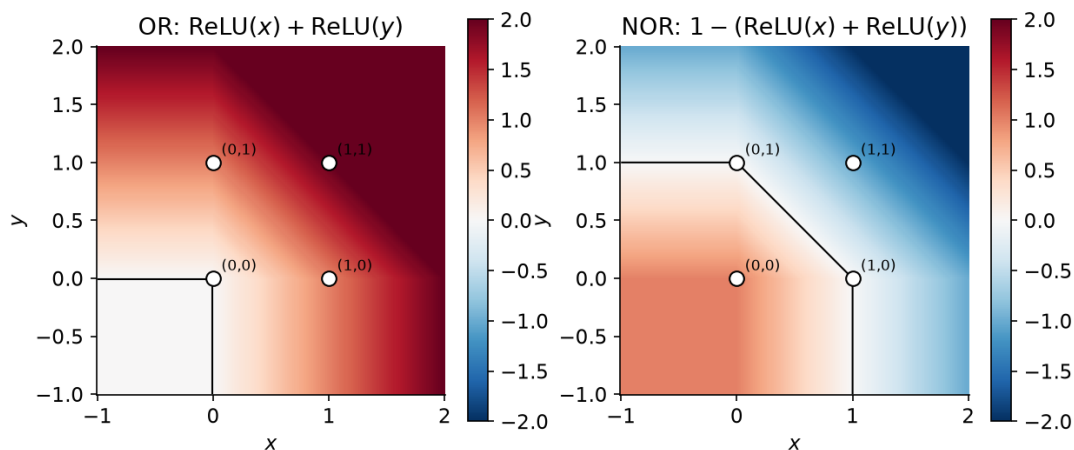
$$n(x) = 1 - x \tag{106}$$

erfüllt die Eigenschaft, NOT zu inter- und extrapolieren. Als neuronales Netz: 1 Neuron, Gewicht $w = -1$, Bias $b = +1$, Aktivierungsfunktion id.

Zurück zu OR

Ansatz: $f(x, y) = x + y$. Leider ist $f(1, 1) = 2 \neq 1$. Besser: $f(x, y) = [x + y > 0]$ – ein Neuron mit Gewichtsvektor $(1, 1)$, Bias $b = 0$ und Aktivierungsfunktion $[\cdot > 0]$ mit Werten in $\{0, 1\}$. Dafür ändern wir die Interpretation leicht zu $(x \leq 0) = \text{F}$, $(x > 0) = \text{T}$. Damit ist der erste Ansatz wieder brauchbar: Dass $f(1, 1) = 2$ herauskommt, stört nun nicht mehr, weil alles über null TRUE heißt. Softere Varianten sind also $f(x, y) := x + y$ oder $\sigma(x + y)$; und noch besser, weil es Negatives sauber abschneidet:

$$f(x, y) := \text{ReLU}(x) + \text{ReLU}(y). \tag{107}$$



OR und NOR als Funktionen auf ganz $\mathbb{R}^2$: Die vier Booleschen Ecken sind markiert, die schwarze Linie ist die Nullstellenmenge.

Komplizierteres aus Einfachem: NOR

Sei $g(x, y) := x \text{ NOR } y$, also $g(F, F) = T$ und sonst F. NOR = NOT OR, also $g(x, y) = n(f(x, y))$ für ein f , welches OR implementiert. Mit $f(x, y) := \text{ReLU}(x) + \text{ReLU}(y)$ und $n(x) := 1 - x$:

$$g(x, y) := 1 - (\text{ReLU}(x) + \text{ReLU}(y)). \quad (108)$$

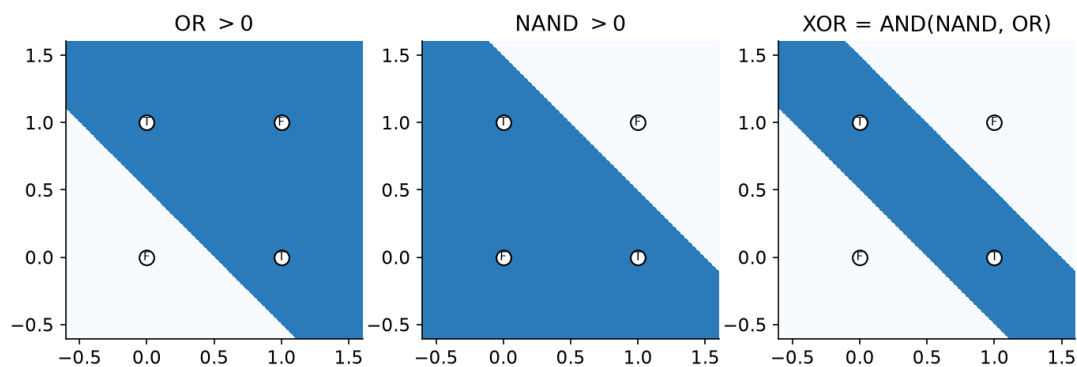
Als Netz: erste Schicht mit **zwei** ReLU-Neuronen, eines pro Eingabe (Gewicht 1, Bias 0), zweite Schicht mit Gewichtsvektor $(-1, -1)$, Bias 1 und id. Ein *einzelnes* Neuron mit Gewichtsvektor $(1, 1)$ täte es nicht: Das berechnet $\text{ReLU}(x+y)$, und das ist etwas anderes als $\text{ReLU}(x) + \text{ReLU}(y)$ – bei $(1, -1)$ etwa 0 gegen 1. Auf den vier Booleschen Ecken fällt der Unterschied nicht auf, außerhalb schon, und genau darum geht es in diesem Kapitel.

Weiter so? XOR

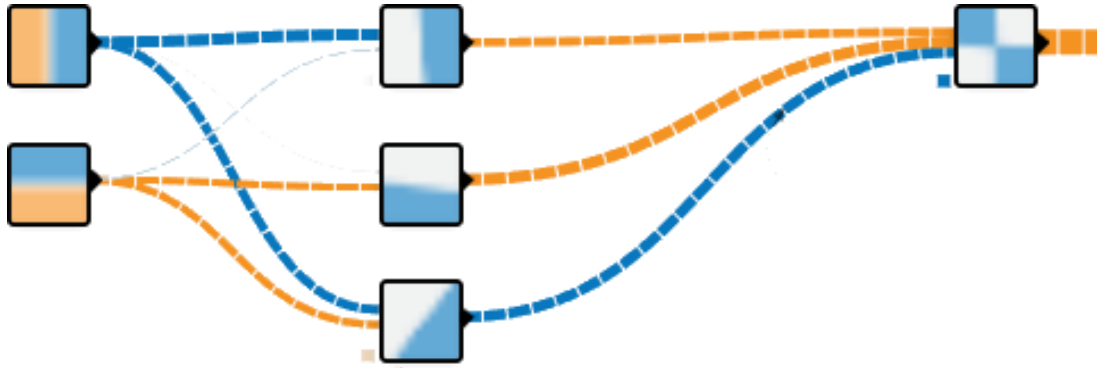
Sei $f(x, y) := x \text{ XOR } y$. Die „Region“ XOR in der Ebene, also die Punkte (x, y) mit $\text{XOR}(x, y) > 0$, ist der Schnitt von $\text{NAND}(x, y) > 0$ mit $\text{OR}(x, y) > 0$:

$$\text{XOR}(x, y) = \text{AND}(\text{NAND}(x, y), \text{OR}(x, y)). \quad (109)$$

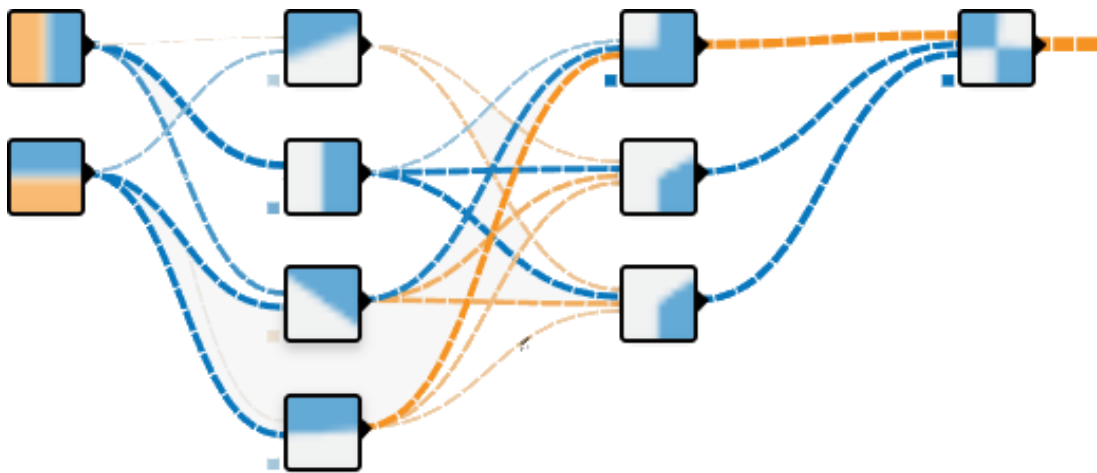
Setzt man das mit den obigen Bausteinen zusammen, so müsste es mit 4 Hidden Layers zu je 4 Neuronen klappen. Gradientenabstieg mit Backpropagation findet allerdings auch Approximationen mit 3 Hidden Layers – und mit 2 Hidden Layers (siehe die beiden Abbildungen unten aus dem TensorFlow Playground).



XOR als Schnitt von NAND und OR.



Vom Gradientenabstieg gefundene Approximation im TensorFlow Playground mit 3 versteckten Schichten.



Dasselbe mit 2 versteckten Schichten.

Take-Home Message

Boolesche Funktionen lassen sich mit neuronalen Netzen darstellen bzw. approximieren, d.h. es gibt Gewichte, sodass die resultierende Funktion tut, was sie soll. Also kann auch Optimierung diese idealen Gewichte finden. Aus elementaren Funktionen lassen sich komplexe verknüpfen; neuronale Netze können aber auch „Abkürzungen“ nehmen und Platz verschwenden (überflüssige Neuronen).

Arithmetik

Wie können wir z.B. $f(x, y) := x^3 + xy^2 - 3y$ mit einem neuronalen Netz berechnen oder approximieren? Einfachere Probleme, die wir dafür zuerst lösen müssen: $f(x, y) := xy$ und $f(x) := x^2$.

Auf $[-3, +3]$ approximieren wir x^2 mit einer stückweise linearen Funktion an ganzzahligen Stützstellen. Die Interpolation durch die Punkte (k, k^2) für $k = -3, \dots, 3$ hat auf $[k, k+1]$ die Steigung $2k+1$, also an jeder inneren Stützstelle einen Knick, bei dem die Steigung um 2 wächst – und jeder Knick ist eine ReLU:

$$Q(x) = 9 - 5 \operatorname{ReLU}(x+3) + \sum_{k=-2}^2 2 \operatorname{ReLU}(x-k). \quad (110)$$

Probe: $Q(0) = 9 - 15 + 2(2+1) = 0$ und $Q(3) = 9 - 30 + 2(5+4+3+2+1) = 9$. Das ist ein Netz mit einer versteckten Schicht aus 6 ReLU-Neuronen (Gewicht 1, Bias $-k$ für $k = -3, \dots, 2$) und einer linearen Ausgabeschicht mit Gewichten $(-5, 2, 2, 2, 2, 2)$ und Bias 9. Die Symmetrie $x^2 = (-x)^2$ haben wir dabei ignoriert – geht das besser? Der Betrag ist selbst ein kleines Netz, $|x| = \operatorname{ReLU}(x) + \operatorname{ReLU}(-x)$ (zwei Neuronen mit Gewichten 1 und -1, dann ein Neuron mit Gewichtsvektor $(1, 1)$), und damit reicht die Interpolation auf $[0, 3]$:

$$P(x) = |x| + 2 \operatorname{ReLU}(|x| - 1) + 2 \operatorname{ReLU}(|x| - 2). \quad (111)$$

Das sind nur noch 3 Neuronen in der Interpolationsschicht – dafür ist das Netz tiefer, weil $|x|$ zwei Schichten vorweg kostet. Weniger Neuronen pro Schicht gegen mehr Schichten: ein Tausch, den wir noch öfter sehen werden.

In der Praxis könnte man PyTorch auch direkt bitten, x^2 oder xy *exakt* zu berechnen (Stichwort *Gates*, etwa in LSTMs), auch wenn man damit das reine MLP verlässt. Der Haken: Im Allgemeinen weiß man vorher nicht, ob man x^2 braucht, um die gesuchte Funktion gut zu approximieren.

Wenn x^2 gut approximiert werden kann, dann auch xy über die Polarisationsidentität:

$$\frac{(x+y)^2 - (x-y)^2}{4} = \frac{x^2 + 2xy + y^2 - x^2 + 2xy - y^2}{4} = xy. \quad (112)$$

Damit setzen wir für eine Approximation Q der Quadratfunktion

$$\operatorname{MUL}(x, y) := \frac{1}{4}Q(x+y) - \frac{1}{4}Q(x-y), \quad (113)$$

und $x+y$ sowie $x-y$ sind lineare Schichten. Mit MUL und Addition bekommen wir alle Polynome, und damit (Weierstraß) alle stetigen Funktionen auf beschränkten Bereichen.

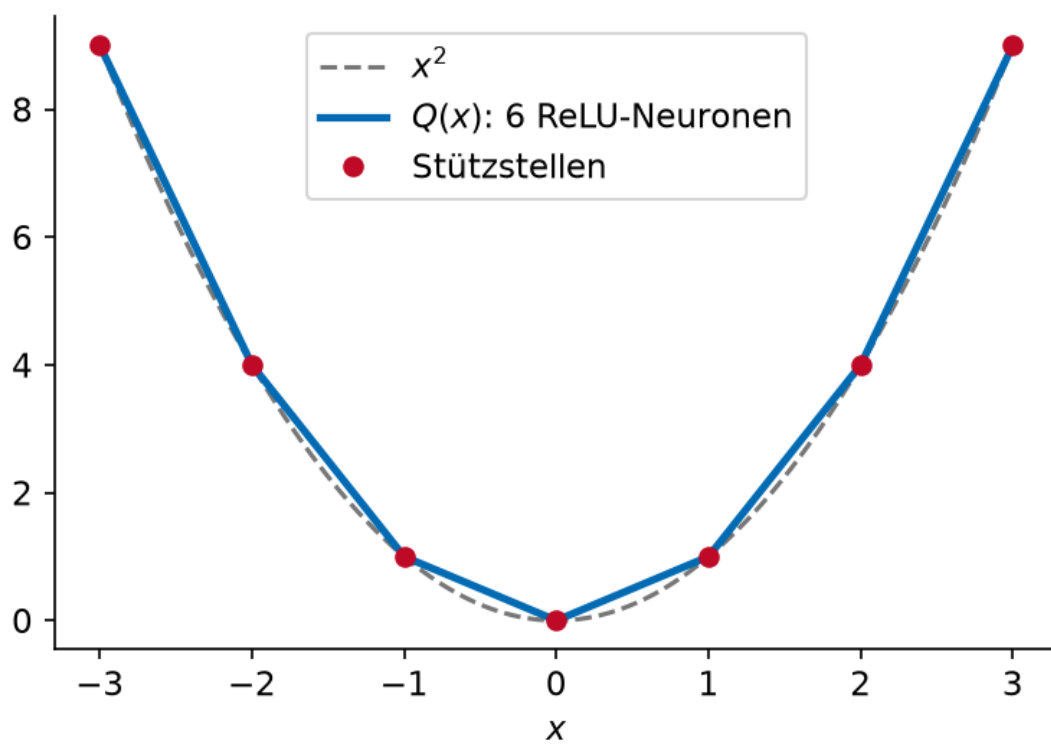
Take-Home Message

Funktionen lassen sich mit neuronalen Netzen darstellen bzw. approximieren. Mit einer Lookup-Table (LUT) und Interpolation sind beliebig gute Näherungen auf *beschränkten* Definitionsbereichen möglich – so rechnet auch die GPU z.B. den Sinus aus. Außerhalb der Stützstellen wird die Approximation beliebig schlecht: **Neuronale Netze interpolieren gut, extrapolieren schlecht.**

3.7.3 Übung

1. Geben Sie Gewichte und Bias für ein einzelnes Perzeptron an, das AND auf $\{0, 1\}^2$ berechnet. Prüfen Sie alle vier Eingaben nach.
2. Bauen Sie NAND aus Ihrem AND, ohne neu zu rechnen. *Tipp: Was macht das Vorzeichenwechseln von w und b mit der Entscheidungsgrenze?*

Antwortprobe: Bei (1) gibt es viele richtige Antworten; alle erfüllen dieselben vier Ungleichungen.



x^2 auf $[-3, 3]$ und die stückweise lineare Interpolation $Q(x)$ an ganzzahligen Stützstellen, ein Netz mit 6 ReLU-Neuronen.

3.8 Universelle Approximation

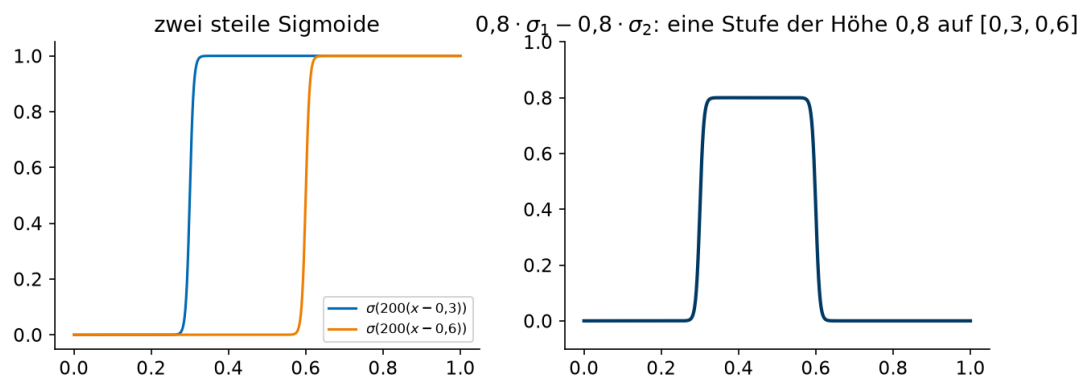
3.8.1 Universelle Approximation

Warum neuronale Netze funktionieren

Universelles Approximationstheorem, grobe Aussage (wir erinnern uns an das [Kapitel „Netze von Hand bauen“](#)): Mit neuronalen Netzen (beliebig viele Neuronen, beliebig viele Layer, fully connected, ReLU) lässt sich jede Funktion beliebig gut approximieren. Der Beweis in fünf Schritten:

1. Jede beschränkte messbare Funktion $f: E \subseteq \mathbb{R} \rightarrow \mathbb{R}$ ist beliebig gut ($\forall \varepsilon > 0$) durch *einfache Funktionen* g, h approximierbar: $g \leq f \leq h$ und $0 \leq h - g < \varepsilon$. Dabei sind einfache Funktionen endliche Linearkombinationen von Indikatorfunktionen messbarer Mengen. (Stetige Funktionen sind integrierbar, integrierbare Funktionen sind messbar.)
2. Eine Funktion $f: E \rightarrow \mathbb{R}$ (für E messbar) ist messbar genau dann, wenn es eine Folge einfacher Funktionen f_n gibt, die punktweise gegen f konvergieren und $|f_n| \leq |f|$ erfüllen. Eine messbare Funktion ist integrierbar, wenn es eine dominierende integrierbare Funktion h gibt, $|f| \leq |h|$. Wir können also sagen: $\{\text{messbare Funktionen}\} \approx \{\text{interessante Funktionen}\}$.
3. Jede einfache Funktion ist beliebig genau approximierbar durch *Stufenfunktionen*, d.h. endliche Linearkombinationen von Indikatorfunktionen von Intervallen. Also: $\{\text{messbare Funktionen}\} \approx \{\text{approximierbar mit Stufenfunktionen}\}$.
4. Eine einzelne Stufe lässt sich mit einem einfachen neuronalen Netz realisieren: Zwei Neuronen mit sehr steilen Sigmoiden an den Intervallgrenzen, mit Gewichten $+h$ und $-h$ in der Ausgabeschicht, ergeben eine Stufe der Höhe h auf dem Intervall (Grafik von Michael Nielsen, Kapitel 4 seines Buchs). Mit ReLUs statt Sigmoiden reichen zwei Neuronen dafür **nicht**: $a \text{ReLU}(x - p) + b \text{ReLU}(x - q)$ ist rechts von beiden Knicken affin mit Steigung $a + b$ und kommt nie auf null zurück – man bekommt eine Rampe mit Plateau, keinen Buckel. Für ein Dreieck über dem Intervall braucht man drei ReLUs, für ein Trapez mit flachem Dach vier.
5. Die leicht zu verstehende Konstruktion mit 2 Hidden Layers bei Michael Nielsen funktioniert auch mit beliebig hoher Ein- und Ausgabedimension, also zur Approximation von Funktionen $\mathbb{R}^n \rightarrow \mathbb{R}^n$. Es gibt auch Konstruktionen mit nur 1 Hidden Layer.

Die richtige Frage ist also nie „Geht das mit neuronalen Netzen?“ (Antwort: JA), sondern „Wie geht das am *effizientesten* mit neuronalen Netzen?“



Zwei steile Sigmoide mit entgegengesetzten Ausgabegewichten ergeben eine Stufe. Aus Stufen baut man Stufenfunktionen, aus Stufenfunktionen approximiert man alles Messbare.

Präzisere Ergebnisse

Theorem 3.1 (Universelle Approximation (Pinkus / Kidger–Lyons)). Sei $E \subseteq \mathbb{R}^d$ eine kompakte (d.h. beschränkte und abgeschlossene) Teilmenge. Sei $\sigma: \mathbb{R} \rightarrow \mathbb{R}$ eine beliebige stetige Funktion, aber keine affin-lineare Abbildung, die in wenigstens einem Punkt stetig differenzierbar mit Ableitung ungleich 0 ist. Dann gibt es zu jeder stetigen Funktion $f: E \rightarrow \mathbb{R}^D$ und jedem $\varepsilon > 0$ ein neuronales Netz n mit Aktivierungen σ und endlich vielen hidden Layers, je höchstens $d + D + 2$ groß, mit

$$\sup_{x \in E} \|n(x) - f(x)\| < \varepsilon. \quad (114)$$

Also: schmale, aber tiefe Netze reichen. Und noch präziser, mit Aussage über die *Größe*:

Theorem 3.2 (Yarotsky (2017)). *Es gibt eine Konstante c , sodass gilt: Zu jeder k -fach stetig differenzierbaren Funktion $f: E \rightarrow \mathbb{R}^D$ und jedem $\varepsilon > 0$ gibt es ein neuronales Netz n mit ReLU-Aktivierungen und $c \log(1/\varepsilon)$ vielen hidden Layers, insgesamt etwa $c \varepsilon^{-d/k}$ Gewichten und $\sup_{x \in E} \|n(x) - f(x)\| < \varepsilon$.*

Siehe auch Andreas Thom, *Approximation of functions by neural networks* ([arXiv:1901.10267](https://arxiv.org/abs/1901.10267), 2019). Die Anzahl der Gewichte wächst mit der Eingabedimension d exponentiell (der Fluch der Dimensionalität), aber glattere Funktionen (größeres k) brauchen weniger – und die Tiefe wächst nur logarithmisch in der Genauigkeit.

Neuronale Netze vs. Bäume und Wälder

Umgekehrt lässt sich jedes ReLU-Netz auch als Entscheidungsbaum schreiben: Jedes ReLU-Neuron ist eine Ja/Nein-Entscheidung „ $\langle w, x \rangle + b > 0$?“, und die Regionen, in denen alle Entscheidungen feststehen, sind genau die Stücke, auf denen das Netz linear ist (Caglar Aytekin, *Neural Networks are Decision Trees*, [arXiv:2210.05189](https://arxiv.org/abs/2210.05189), 2022). Der Baum ist allerdings exponentiell groß.

3.8.2 Übung

1. Geben Sie zwei Neuronen mit steilen **Sigmoiden** und Ausgabegewichte an, deren Summe die Funktion $f(x) = 1$ für $x \in [1, 2]$ und 0 sonst **näherungsweise** darstellt. *Tipp: Zwei steile Stufen an den Stellen 1 und 2, die zweite mit umgekehrtem Vorzeichen. Der Steilheitsparameter h bestimmt, wie gut die Näherung ist.*
2. Mit **ReLU**s statt Sigmoiden reichen zwei Neuronen nicht. Wie viele brauchen Sie für ein Dreieck über $[1, 2]$, wie viele für ein Trapez mit flachem Dach? *Tipp: Sehen Sie sich an, welche Steigung die Summe rechts von allen Knicken hat, und was sie sein müsste, damit die Funktion wieder auf 0 zurückkommt.*
3. Wie viele Neuronen brauchen Sie für eine Treppenfunktion mit k Stufen?

Antwortprobe: Die beiden Zahlen in (2) unterscheiden sich um genau eins, und die Antwort auf (3) ist linear in k .

Weiterlesen

Prince, *UDL*, Kap. 3 (Universal approximation theorem), Kap. 4 (Depth vs. width) und Kap. 20 (Why does deep learning work?). Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).

4 Representation Learning

4.1 Autoencoder

4.1.1 Lernen ohne Label

Im Kontrast zu unserem bisherigen Vorgehen

Bisher haben wir ein Neuron oder ein neuronales Netz zur Klassifikation oder Regression verwendet: Eine nicht bekannte Funktion f , für die wir manche Werte kennen (Label $y = f(x)$ für Trainingsdaten (x, y)), wollen wir mit dem Modell „schätzen“. Was, wenn wir keine Label haben?

Herleitung zur abstrakten Idee

Ohne Label müssen wir den Output des neuronalen Netzes anders bewerten. Wie Erfolg messen? Welches Ziel ist im Training noch sinnvoll?

Ein mögliches Ziel: **Transformation der Eingabe in ein anderes Koordinatensystem** (Feature Extraction). Das kann man rückgängig machen – und messen, wie gut das klappt. Abstrakt: f ist nicht bekannt, aber $g \circ f$ ist bekannt, nämlich $g(f(x)) = x$. Hier heißt f **Encoder** und g **Decoder**, und $g \circ f$ soll die Identität sein.

Das allein wäre trivial ($f = g = \text{id}$). Interessant wird es mit einer Einschränkung: **Dimensionsreduktion**. Versuche, die Dimensionalität zu reduzieren ohne Informationsverlust – transformiere die Eingabe in ein *kleineres* Koordinatensystem, sodass man es trotzdem rückgängig machen kann. Der Encoder $f: \mathbb{R}^n \rightarrow \mathbb{R}^k$ mit $k \ll n$ muss dann lernen, was an den Daten wesentlich ist.

Loss-Funktion

Trainingsziel: Output so ähnlich wie möglich zum Input (**Rekonstruktion**). Vergleiche Input mit Output, z.B. über die mittlere quadratische Abweichung der Pixelwerte:

$$L(x; f, g) = \frac{1}{N} \sum_{i=1}^N \|x_i - g(f(x_i))\|^2. \quad (115)$$

Der Autoencoder ist also ein neuronales Netz mit einem „Flaschenhals“ in der Mitte, trainiert mit MSE zwischen Ein- und Ausgabe – und die Label sind die Eingaben selbst. Man nennt das auch *self-supervised*.

Im Vergleich zur Dimensionsreduktion mit PCA: Ist der Autoencoder linear (keine Aktivierungsfunktionen), so findet er denselben Unterraum wie PCA. Mit Nichtlinearitäten kann er gekrümmte Strukturen in den Daten erfassen, die PCA verfehlt, und rekonstruiert z.B. Bilder bei gleicher Dimension des Flaschenhalses deutlich besser.

4.1.2 Autoencoder für MNIST

Das Autoencoder-Layout

Noch einmal genauer: Ein Autoencoder ist eine Regression mit Input x , einem Vektor der Länge D . Der **Encoder** reduziert in mehreren Schichten von Dimension D auf Dimension L , der **Decoder** erhöht in mehreren Schichten von Dimension L zurück auf Dimension D . Input und Output liegen im „Datenraum“ (bei Bildern: „Pixelraum“, pixel space); die Flaschenhals-Schicht der Dimension L ist der **latente Raum** (latent space).

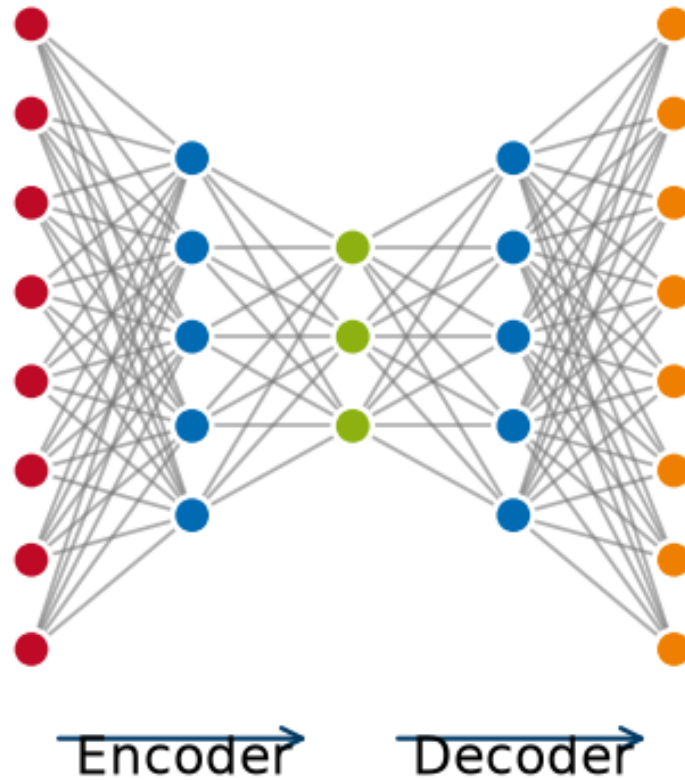
Anwendungsbeispiel: MNIST

MNIST besteht aus 28×28 Graustufen-Bildern handgeschriebener Ziffern 0–9, mit Label; 60 000 Trainings- und 10 000 Testbilder. Mögliche Tasks:

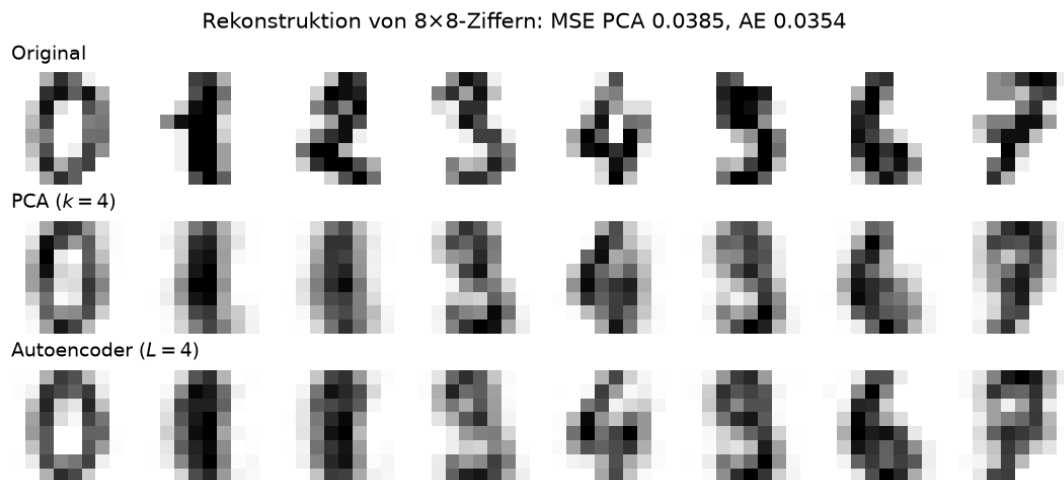
- Unsupervised: Kompression der Bilder. Autoencoder = Encoder + Decoder, der Encoder ist der Kompressor.
- Unsupervised: neue Bilder generieren. Die Verteilung der Punkte im latenten Raum modellieren, sampeln, dekodieren – dazu mehr im nächsten Kapitel (VAE).
- Supervised: Klassifikation.
- Supervised: neue Bilder mit vorgegebener Ziffer generieren.

Zunächst: Kompression und Klassifikation, mit den Zielen *Accuracy erhöhen* und *Overfitten vermeiden*. Variationsmöglichkeiten dafür gibt es an drei Stellen: die Daten modifizieren (putzen?), die Modellarchitektur (tiefer?), und die Trainings-Hyperparameter (wie genau wird gelernt?).

Input $x \in \mathbb{R}^D$ → Latenter Vektor $z \in \mathbb{R}^L$ → Rekonstruktion $\hat{x}$



Autoencoder schematisch: Encoder, Flaschenhals (latenter Vektor), Decoder.



Rekonstruktion von 8×8-Ziffern aus scikit-learn durch PCA mit 4 Komponenten und durch einen kleinen Autoencoder mit $L = 4$ (erzeugt mit `sklearn.neural_network.MLPRegressor`).

Code-Beispiel: Modellarchitektur

Ein Autoencoder in PyTorch ist ein `nn.Module` mit zwei `nn.Sequential`-Blöcken. Schematisch:

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

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
IMAGESIZE = 28

class Autoencoder(nn.Module):
    def __init__(self, latent_dim=12):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(IMAGESIZE**2, 256), nn.ReLU(True),
            nn.Linear(256, 64), nn.ReLU(True),
            nn.Linear(64, latent_dim),
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 64), nn.ReLU(True),
            nn.Linear(64, 256), nn.ReLU(True),
            nn.Linear(256, IMAGESIZE**2), nn.Sigmoid(),
        )

    def forward(self, x):
        latent = self.encoder(x)
        return latent, self.decoder(latent)
```

Das Sigmoid am Ende des Decoders sorgt dafür, dass die Ausgabe wie Pixelwerte in $[0, 1]$ liegt. Man kann auch dem Encoder ein Sigmoid ans Ende setzen; nötig ist das erst für den Sparse Autoencoder weiter unten, dessen Strafterm latente Werte in $[0, 1]$ voraussetzt. Die Architektur lässt sich mit [Netron](#) visualisieren, nachdem man das Modell mit `torch.save(model, 'model.pth')` gespeichert hat.

Code-Beispiel: Training

```
transform_mnist = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1)), # 28x28 -> 784
])
mnist_data = datasets.MNIST(root='./data', train=True, download=True,
                             transform=transform_mnist)
data_loader = DataLoader(dataset=mnist_data, batch_size=512,
                          num_workers=8, shuffle=True)

model = Autoencoder(latent_dim=12).to(device)
reconstruction_criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=1e-2)

for epoch in range(15):
    for img, labels in data_loader:
        img = img.to(device)
        latent, reconstructed = model(img)
        reconstruction_loss = reconstruction_criterion(reconstructed, img)
        loss = reconstruction_loss
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Neu gegenüber dem [Kapitel „Tensoren“](#) sind **Dataset** und **DataLoader**: Der Datensatz liefert einzelne Samples (hier mit einer `transform`, die das Bild in einen Vektor der Länge 784 verwandelt), der Loader

fasst sie zu zufällig gemischten Mini-Batches zusammen. Die Trainingsschleife selbst ist dieselbe wie immer, nur dass das Label das Bild selbst ist.

Tricks beim Training

- **Mehr Epochen trainieren!** – aber nicht beliebig viele: Wenn der Loss auf dem Testset wieder *steigt*, während er auf den Trainingsdaten weiter sinkt, overfittet das Modell. **Early Stopping** heißt, an diesem Punkt aufzuhören (bzw. die Gewichte von dort zu behalten).
- **Data Augmentation:** die Trainingsdaten durch zufällige Transformationen vervielfachen, die das Label nicht ändern – Bilder leicht drehen, verschieben, spiegeln, verrauschen. Das Modell sieht nie zweimal dasselbe Bild und kann es nicht auswendig lernen; auch das ist Regularisierung.
- **Lernrate anpassen** – statt `lr=1e-3` mal mit `lr=1e-2` probieren.
- Die Gewichte können zu groß werden, dann kann man eine **L2-Regularisierung** nutzen, den *Weight Decay* (als Parameter für den Optimierer, wie die Lernrate): `weight_decay=1e-5`. Das addiert $\lambda \|W\|^2$ zum Loss und zieht die Gewichte bei jedem Schritt ein wenig Richtung null.
- SGD kann sehr langsam sein. Es gibt eine ganze Reihe von Tricks wie z.B. eine adaptive Lernrate und cleveres Mitteln über mehrere Batches hinweg, die in **Adam** implementiert sind: `optim.Adam(model.parameters(), lr=1e-3)`. Faustregel für den Rest des Kurses: Adam statt SGD, ohne dass wir Optimierer weiter vertiefen. Was Adam genau tut, steht im [Kapitel „Optimierer“](#).
- Oft ist I/O oder die CPU ein Bottleneck beim Laden der Daten. Im Konstruktor des **DataLoader** mit anderen `batch_sizes` experimentieren (im Zweifel mehr, bis Out of Memory), und mehrere CPU-Kerne zum Laden der Daten nutzen: `num_workers=8`.

4.1.3 Label Autoencoder

Supervised: Klassifikator

Ein „klassischer“ Klassifikator-Ansatz auf Basis des Autoencoders: Encoder vom Autoencoder, dann ein linearer Layer auf 10 Dimensionen. Dessen Ausgabe sind die **Logits**; erst ein Softmax darauf macht daraus Wahrscheinlichkeiten, und ein `argmax` dahinter die „harte“ Klassifikation.

Loss-Funktion: **Negative Log-Likelihood** = Kreuzentropie $\hat{=}$ Kullback-Leibler-Divergenz zwischen echten Labels und Vorhersage (siehe [Kapitel „Loss-Funktionen und Maximum Likelihood“](#)). Das ist der *classification loss*, im Gegensatz zum *reconstruction loss*. In PyTorch: `nn.CrossEntropyLoss()`, das Softmax und NLL in einem Schritt (numerisch stabiler) berechnet und direkt auf die Logits angewendet wird. **Deshalb gehört das Softmax nicht ins Modell**, wenn man diese Loss-Funktion benutzt – sonst wird zweimal normiert, die Verteilung wird flach und das Training langsam. Erst bei der Inferenz, wenn man Wahrscheinlichkeiten sehen will, wendet man `torch.softmax` von Hand an.

Label Autoencoder

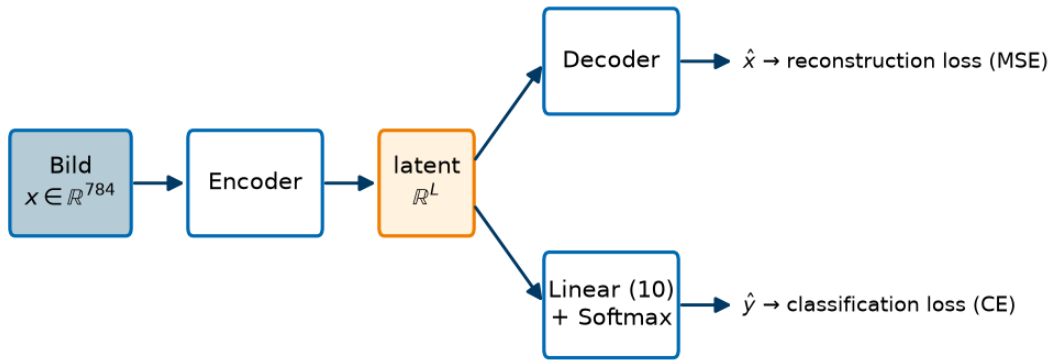
Warum nicht beides? Encoder vom Autoencoder, darauf der lineare Layer auf 10 Dimensionen (Logits) – und *daneben* der Decoder vom Autoencoder. Input ist ein Bild, Output sind Rekonstruktion und Label-Vorhersage, und die Loss-Funktion ist

$$L = \text{reconstruction_loss} + \text{classification_loss}. \quad (116)$$

Der Encoder muss jetzt einen latenten Raum finden, der *beides* kann: das Bild rekonstruieren und die Ziffer erkennen. Die Rekonstruktion wirkt dabei wie eine Regularisierung des Klassifikators – der latente Raum darf nichts Wichtiges wegwerfen.

Tricks bei der Modellarchitektur

- **Mehr Layer!**
- Dem Problem sterbender Neuronen entgegenwirken mit anderen Aktivierungsfunktionen als ReLU, z.B. dem **Swish** (in PyTorch `nn.SiLU`).
- **Dropout-Layer:** Zufällig ein paar Einträge des eingehenden Vektors auf 0 setzen (`nn.Dropout(0.3)` setzt 30 % auf null, nur während des Trainings). Verhindert zu starke Konzentration auf einige wenige Neuronen und wirkt gegen Overfitting.



Label Autoencoder: ein Encoder, zwei Köpfe, zwei Loss-Terme.

- **Batch-Normalization-Layer** (`nn.BatchNorm1d`): Nach jedem Batch den Output des vorigen Layers auf eine Normalverteilung normieren, d.h. Mittelwert abziehen und durch die Standardabweichung teilen, beides über den Batch berechnet. Die Transformation (Skalierung und Translation) auf eine andere Normalverteilung als die Standardnormalverteilung sind erlernbare Gewichte dieses Layers. Theoretisch etwas kompliziert (wieso es nützt), praktisch sehr nützlich.

4.1.4 Autoencoder-Varianten

Zur Erinnerung: Ein Autoencoder besteht aus Encoder $\mathbb{R}^D \rightarrow \mathbb{R}^L$ und Decoder $\mathbb{R}^L \rightarrow \mathbb{R}^D$, mit dem latenten Raum der Dimension L als Flaschenhals. In diesem Abschnitt variieren wir die Loss-Funktion und die Eingabe, und sehen, was das mit dem latenten Raum macht.

Sparse Autoencoder

Idee: die Loss-Funktion um **Strafterme** ergänzen. Hier: dafür sorgen, dass ein latenter Vektor *dünn besetzt* (sparse) ist, also die meisten Einträge nahe 0 sind.

Lösung: Unter der Annahme, dass die latenten Vektoren Einträge in $[0, 1]$ haben (z.B. durch ein Sigmoid am Ende des Encoders), betrachten wir für jede latente Dimension j die mittlere Aktivierung $\hat{\varrho}_j$ über den Batch und bestrafen die Kullback-Leibler-Divergenz zwischen der Bernoulli-Verteilung mit Wahrscheinlichkeit $\varrho = 0,1$ und der Bernoulli-Verteilung mit Wahrscheinlichkeit $\hat{\varrho}_j$:

$$L_{\text{sparse}} = \sum_{j=1}^L D_{KL}(\text{Ber}(\varrho) \parallel \text{Ber}(\hat{\varrho}_j)) = \sum_{j=1}^L \left(\varrho \ln \frac{\varrho}{\hat{\varrho}_j} + (1 - \varrho) \ln \frac{1 - \varrho}{1 - \hat{\varrho}_j} \right). \quad (117)$$

Der Term ist 0 genau dann, wenn jede latente Dimension im Mittel nur zu 10 % „an“ ist. Im Code muss er ausdrücklich zum Rekonstruktions-Loss addiert werden, `loss = reconstruction_loss + beta * sparsity_loss`; eine Funktion, die ihn nur berechnet, ändert am Training nichts. Dense latents nutzen alle Dimensionen für jeden Datenpunkt; sparse latents verteilen die Datenpunkte auf wenige aktive Dimensionen, was die einzelnen Dimensionen interpretierbarer macht.

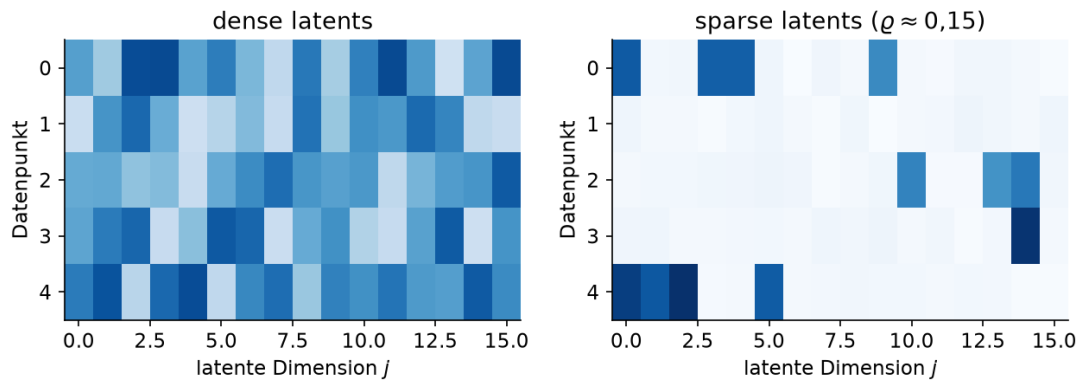
Denoising Autoencoder

Idee: Daten nicht putzen, sondern *verschmutzen*. Noise ε generieren, auf die Daten addieren, $x_{\text{noisy}} = x + \varepsilon$, und das als Input für den Autoencoder verwenden. Die Loss-Funktion vergleicht aber $\text{dec}(\text{enc}(x_{\text{noisy}}))$ mit dem *sauberen* x :

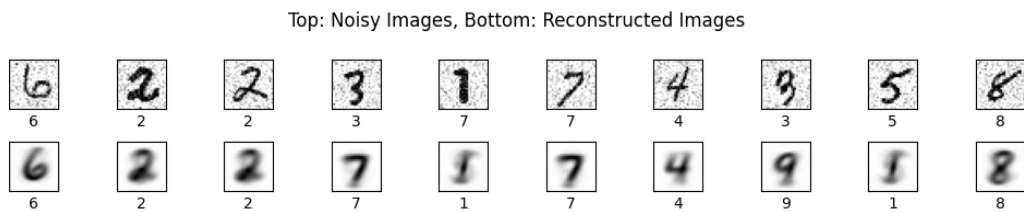
$$L_{\text{denoise}} = \text{MSE}(x, \text{dec}(\text{enc}(x + \varepsilon))). \quad (118)$$

Das zwingt das Netzwerk dazu, einzelne Pixel nicht zu stark zu gewichten und eher globale Strukturen zu suchen – ein einzelner verrauschter Pixel darf die Rekonstruktion nicht kippen.

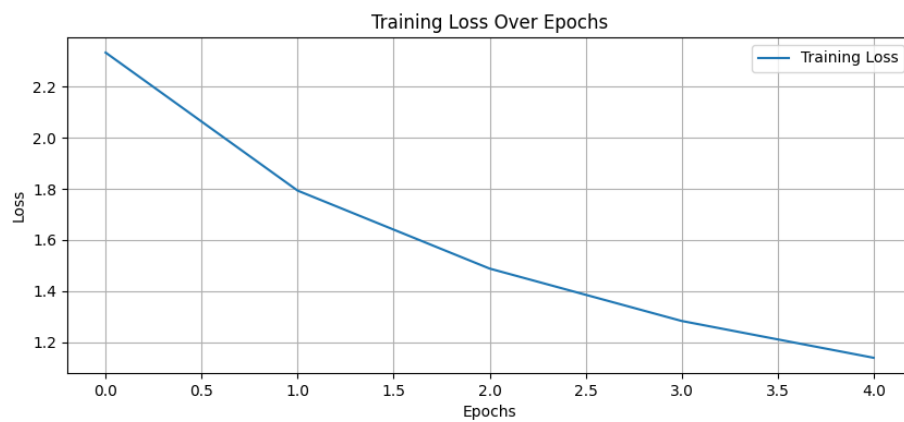
Die drei Abbildungen oben stammen aus einem *Denoising Sparse Label Autoencoder* – alle Tricks auf einmal, mit SiLU, BatchNorm und Dropout: verrauschte Eingabe hinein, sauberes Bild und Ziffer heraus, dazu ein Sparsity-Strafterm auf dem latenten Vektor.



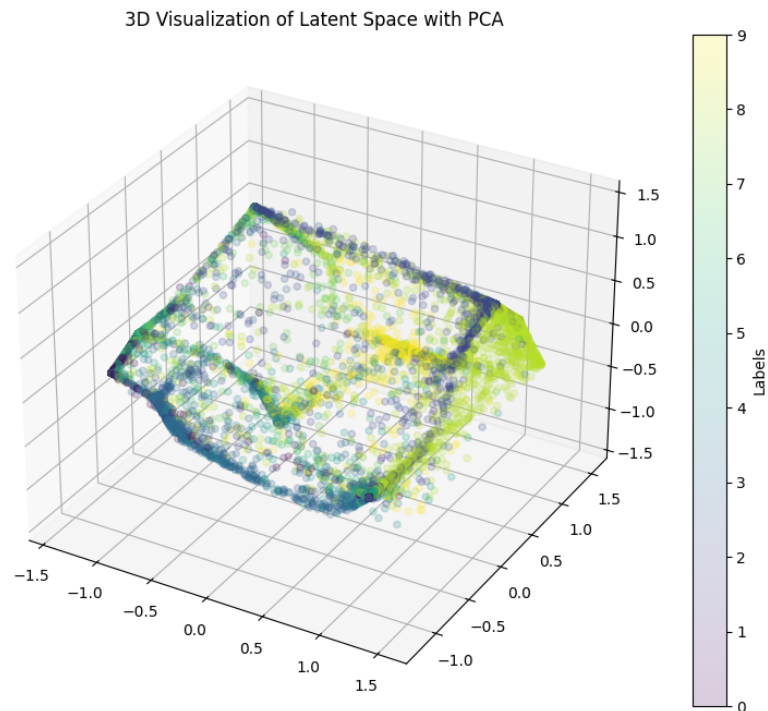
Latente Vektoren von fünf Datenpunkten (Zeilen), 16 Dimensionen (Spalten): dicht besetzt vs. dünn besetzt.



Denoising Sparse Label Autoencoder auf MNIST (Musterlösung zu einem Übungsblatt, WiSe 2025/26): verrauschte Eingabe, Rekonstruktion, Original.



Dasselbe Modell: Loss-Kurve über die Epochen.



Der latente Raum desselben Modells, per PCA auf zwei Dimensionen projiziert und nach Ziffer eingefärbt.

Bisher, und was es bringt

Wir haben das Autoencoder-Design eingesetzt, um Bilder zu komprimieren (mit Sparsity Constraint), Bilder zu klassifizieren, Bilder zu entzaubern – und all das zusammen. Ein ML-System, welches diese Bilder verarbeiten soll, kann nun den fertig trainierten Encoder verwenden und direkt auf den Repräsentationen im Latent Space arbeiten. Das ist **Modularisierung im Training**: Der Encoder wird einmal trainiert und dann als Feature-Extraktor wiederverwendet.

Frage: Welche Struktur finden wir im Latent Space bzw. in den Latent Vectors der Daten? Die **Latent-Manifold-Hypothese** besagt, dass die Daten in $\mathbb{R}^D$ nahe einer niedrigdimensionalen gekrümmten Fläche (Mannigfaltigkeit) liegen und der latente Raum diese Fläche „flach“ parametrisiert. Dazu mehr im [Kapitel „Latente Räume geometrisch“](#).

Später werden wir das Autoencoder-Design einsetzen, um Text (Strings) natürlicher Sprache zu kodieren, sodass die latente Repräsentation interessante Struktur aufweist – das ist word2vec ([Kapitel „word2vec“](#)).

4.1.5 Übung

Ein Autoencoder für MNIST: $784 \rightarrow 128 \rightarrow 32 \rightarrow 128 \rightarrow 784$, alle Layer voll verbunden mit Bias.

1. Wie viele Parameter hat er? Wie viele davon im Encoder? *Tipp: Ein Layer $n \rightarrow m$ hat $n \cdot m + m$ Parameter.*
2. Beim Denoising-Autoencoder: Auf welche der beiden Größen – Eingabe oder Zielbild – kommt das Rauschen, und was wäre der Effekt, wenn man es auf die andere legte?

Antwortprobe: Bei (1) liegt das Ergebnis zwischen 200 000 und 250 000.

Weiterlesen

Prince, *UDL*, Kap. 14 (Unsupervised learning) als Einordnung; Deisenroth et al., *MML*, Kap. 10 (PCA) für den linearen Fall. Beide Bücher sind frei verfügbar, siehe [Über dieses Skript](#).

4.2 Variational Autoencoder

4.2.1 Vom Autoencoder zum VAE

Motivation: probabilistische Modelle

Beim klassischen Autoencoder bildet der Encoder einen Datenpunkt x aus dem Datenraum $\mathbb{R}^D$ auf einen Punkt $\text{enc}(x)$ im latenten Raum $\mathbb{R}^L$ ab, der Decoder einen Punkt $y \in \mathbb{R}^L$ zurück in den Datenraum (D und L sind dieselben Dimensionen wie im [Kapitel „Autoencoder“](#)); Ziel ist $\text{MSE}(x, \text{dec}(\text{enc}(x)))$ möglichst klein. Die Modellannahme war aber: Die Punkte $x \in D$ sind eine Stichprobe aus einer *Verteilung*. Können wir die Verteilung direkt modellieren, in $\text{enc}(x)$?

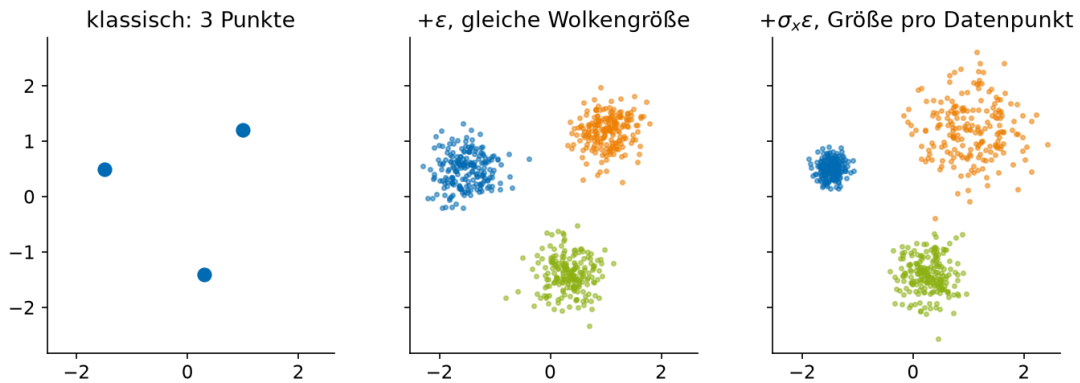
Variation des Ziels: Auch $\text{MSE}(x, \text{dec}(\text{enc}(x) + \varepsilon))$ soll möglichst klein sein, für einen normalverteilten Fehler $\varepsilon \sim \mathcal{N}(0, 1)$. **Variation des Trainings:** Nach $\text{enc}(x)$ ein ε sampeln, addieren, dann dec. Der Decoder muss dann nicht nur den Punkt $\text{enc}(x)$, sondern eine ganze Umgebung davon auf x abbilden – der latente Raum wird „glatt“.

Oh: Wie geht Backpropagation für „ein ε sampeln“? Da ε nicht von den Gewichten abhängt, ganz normal mit der Kettenregel:

$$\frac{d}{dw_{\text{enc}}} \text{dec}(\text{enc}(x) + \varepsilon) = \text{dec}'(\text{enc}(x) + \varepsilon) \cdot \frac{d}{dw_{\text{enc}}} \text{enc}(x). \quad (119)$$

Das gesampelte ε ist im Rückwärtsdurchlauf einfach eine Konstante.

Beispiel im latenten Raum. Mit 3 Datenpunkten: Klassisch sind das 3 Punkte. Mit $+\varepsilon$ werden es 3 Wolken gleicher Größe. Mit $+\sigma\varepsilon$ für einen Skalierungsfaktor σ kann man die Wolkengröße steuern. (In diesem Kapitel ist σ durchgehend eine Streuung, nicht die logistische Sigmoidfunktion aus dem [Kapitel „Logistische Regression“](#).) Mit 300 Datenpunkten: Klassisch eine Punktwolke mit Löchern und Ausreißern; gut wäre eine Verteilung, die den latenten Raum lückenlos und gleichmäßig füllt, sodass man an *jeder* Stelle dekodieren kann.



Drei Datenpunkte im latenten Raum – klassisch, mit gleichem Rauschen $+\varepsilon$, mit datenabhängigem Rauschen $+\sigma_x \varepsilon$.

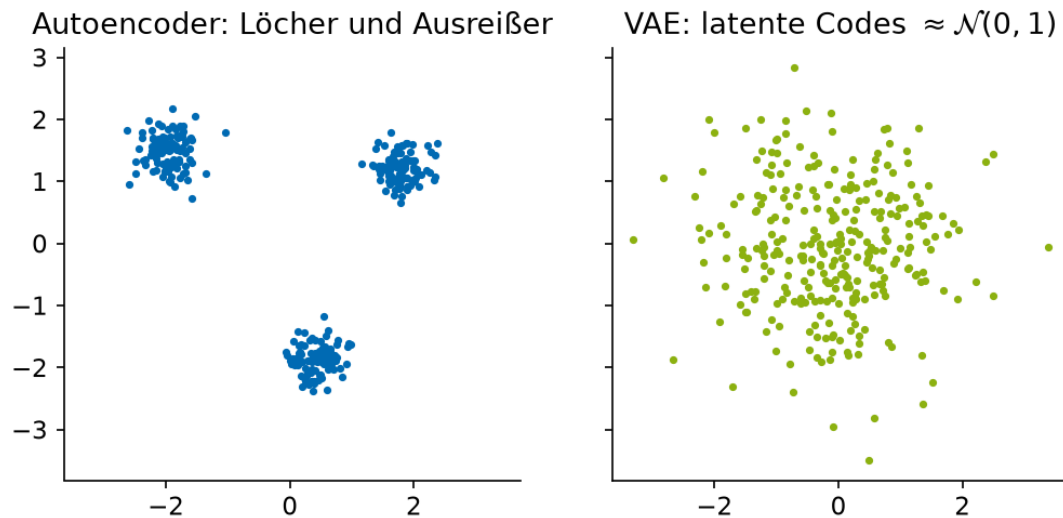
Modelliere die Verteilung also mit einer gaußschen Mischung im latenten Raum, in der die Störung ε von den Daten abhängt: Zu jedem Datenpunkt $x \in D$ brauchen wir zusätzlich zu $\text{enc}(x) \in L$ auch eine Standardabweichung σ_x (bzw. besser noch eine Kovarianzmatrix $\text{cov}(x)$), dann $\text{dec}(\text{enc}(x) + \varepsilon)$ mit $\varepsilon \sim \mathcal{N}(0, \sigma_x^2)$. Aber Sampling aus einer Verteilung, deren Parameter σ_x von den Gewichten abhängen, lässt sich nicht ableiten. **Reparametrisierungstrick:** statt $\varepsilon \sim \mathcal{N}(0, \sigma_x^2)$ besser

$$y := \text{enc}(x) + \sigma_x \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 1), \quad (120)$$

denn dann ist ε wieder eine Konstante und σ_x ein ganz normaler Ausgang des Encoders, durch den Gradienten fließen.

Training

Wie trainiert man einen Variational Autoencoder? Für jeden Datenpunkt $x \in D$ berechne $\text{enc}(x) = (\mu_x, \sigma_x)$ – der Encoder hat jetzt $2L$ Ausgänge –, generiere ein $\varepsilon \sim \mathcal{N}(0, 1)$ und setze $y := \mu_x + \sigma_x \varepsilon$. Die Loss-Funktion ist *naiv* $\text{MSE}(x, \text{dec}(y))$. Das hat einen Haken, und zwar in beide Richtungen: Das Netz kann $\sigma_x \rightarrow 0$ lernen und ist dann wieder ein klassischer Autoencoder. Oder es lässt μ_x und σ_x beliebig



300 Datenpunkte – ein klassischer Autoencoder lässt Löcher und Ausreißer, ein VAE zieht die Codes zur Standardnormalverteilung.

groß werden, dann explodieren die latenten Vektoren, die Wolken der einzelnen Datenpunkte liegen weit auseinander, und der latente Raum füllt sich nie lückenlos.

Stabileres Training mit dem Extra-Term **Variational Loss**:

$$L_{\text{VAE}} = \text{MSE}(x, \text{dec}(y)) + D_{\text{KL}}(q(y | x) \| \mathcal{N}(0, 1)), \quad (121)$$

wobei $q(y | x) = \mathcal{N}(\mu_x, \sigma_x^2)$ die Dichte der latenten Verteilung für ein festes x ist. Dabei sind die beiden Standardnormalverteilungen – das $\mathcal{N}(0, 1)$ für ε und das $\mathcal{N}(0, 1)$ im KL-Term – nicht „dieselben“!! Die erste ist die Rauschquelle, die zweite ist die *Prior*-Verteilung, an die sich die Gesamtheit der latenten Verteilungen anschmiegen soll, damit der latente Raum um den Ursprung herum lückenlos gefüllt ist. In der Praxis lässt man den Encoder statt σ_x den Logarithmus der Varianz ausgeben, $\ell_x := \ln \sigma_x^2$ (im Code meist **logvar**), damit kein Ausgang auf positive Werte eingeschränkt werden muss; dann ist $\sigma_x = \exp(\ell_x/2)$. Für Normalverteilungen hat der KL-Term eine geschlossene Form:

$$D_{\text{KL}}(\mathcal{N}(\mu, \sigma^2) \| \mathcal{N}(0, 1)) = \frac{1}{2} (\mu^2 + \sigma^2 - 1 - \ln \sigma^2). \quad (122)$$

Diese Loss-Funktion ist der negative **ELBO** (evidence lower bound) aus dem Kapitel [Expectation Maximization](#) in Data Science – bis auf einen Faktor und eine additive Konstante, die daher rühren, dass wir den Rekonstruktionsterm als schlichten MSE schreiben statt als negative Log-Likelihood eines gaußschen Decoders mit fester Varianz. Für die Optimierung macht das keinen Unterschied, für den Vergleich der Zahlenwerte schon.

Zwei Fragen, die immer kommen

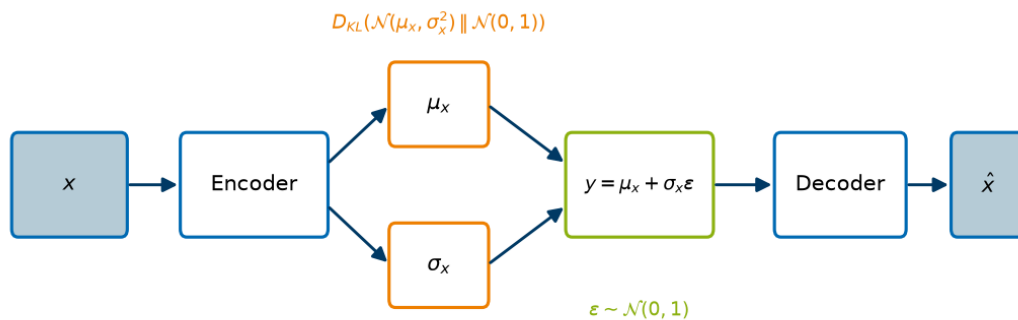
Wie wird σ_x berechnet? Gar nicht aus den Daten, sondern erfunden: μ_x und σ_x sind einfach die $2L$ Ausgänge des Encoders, und der Gradientenabstieg stellt sie so ein, dass der Loss klein wird. Der Encoder lernt, wie unsicher er sich bei jedem Datenpunkt sein darf.

Was hat das mit PCA und EM zu tun? Ein linearer Autoencoder ohne Aktivierungsfunktionen findet denselben Unterraum wie PCA ([Kapitel „Autoencoder“](#)), nur bis auf Drehung und Vertauschung der Achsen. Ein Gaussian Mixture Model mit EM aus Data Science fittet *wenige* Cluster, jeder mit eigener Normalverteilung, und ordnet jeden Datenpunkt einem Cluster zu. Der VAE geht den umgekehrten Weg: *Jeder* Datenpunkt bekommt seine eigene Normalverteilung im latenten Raum, und der KL-Term sorgt dafür, dass alle zusammen ungefähr eine Standardnormalverteilung ergeben.

Recap

Ein VAE ist ein Autoencoder, dessen Encoder statt eines Punkts eine Normalverteilung im latenten Raum ausgibt, dessen Decoder aus einer Stichprobe daraus rekonstruiert, und dessen Loss die Rekon-

struktion plus die KL-Divergenz zum Standardnormal-Prior ist. Danach kann man neue Daten generieren, indem man $y \sim \mathcal{N}(0, 1)$ sampelt und dekodiert. Schön zusammengefasst in Lilian Wengs Blogpost [From Autoencoder to Beta-VAE](#). VAEs sind heute noch eine Komponente der Bildgeneratoren (DALL-E, Midjourney, Stable Diffusion): Die Diffusion findet im latenten Raum eines VAE statt.



Der Variational Autoencoder mit Reparametrisierungstrick: Der Encoder liefert μ_x und σ_x , das Rauschen ϵ ist eine Konstante für die Backpropagation, der KL-Term zieht die Codes zum Prior.

4.2.2 Übung

Der variational loss term für eine latente Dimension ist $\frac{1}{2} (\mu^2 + \sigma^2 - \ln \sigma^2 - 1)$.

1. Für welches Paar (μ, σ) wird er null? Rechnen Sie nach, dass er dort ein Minimum hat. *Tipp: Leiten Sie nach μ und nach σ getrennt ab.*
2. Berechnen Sie den Wert für $(\mu, \sigma) = (2, 1)$ und für $(0, 0.5)$. Welche der beiden Abweichungen bestraft der Term stärker?
3. Der Reparametrisierungstrick schreibt $z = \mu + \sigma\epsilon$ mit $\epsilon \sim \mathcal{N}(0, 1)$. Warum lässt sich ohne ihn nicht nach μ und σ ableiten?

Antwortprobe: Bei (2) ist genau ein Wert größer als 1.

Weiterlesen

Prince, *UDL*, Kap. 17 (Variational autoencoders) mit ELBO und Reparametrisierung; Deisenroth et al., *MMML*, Kap. 11 (Gaussian mixture models, EM) zum Vergleich. Beide Bücher sind frei verfügbar, siehe [Über dieses Skript](#).

4.3 Wortvektoren: word2vec

4.3.1 word2vec

Von der Anwendung her gedacht

Ein Beispiel-Task, unsupervised mit neuronalen Netzen: Alle Dokumente D_i einlesen als Liste von Wörtern (getrennt an Leerzeichen, Satzzeichen entfernt), dann jedes einzelne Wort mit einem zuvor trainierten neuronalen Netz umrechnen in eine andere Repräsentation, die sich besser vergleichen lässt. Die andere Repräsentation verschaffen wir uns durch latente Vektoren aus einem Autoencoder. Das ist die Idee von **word2vec** (Mikolov et al., Google, 2013).

Vorverarbeitung und Ziel

1. Wir zählen alle Wörter (bzw. Token) im Korpus,
2. entfernen Stopwords und sortieren nach Häufigkeit,
3. behalten die häufigsten N Wörter (z.B. $N = 5000$),
4. kodieren diese mit One-Hot-Encoding als Vektoren in $\mathbb{R}^N$,
5. mit dem Ziel, eine Abbildung $e: \mathbb{R}^N \rightarrow \mathbb{R}^K$ zu finden, $K \ll N$ (z.B. $K = 200$),
6. mit einer „Decoder“-Abbildung $d: \mathbb{R}^K \rightarrow \mathbb{R}^N$, sodass $d(e(w)) = w$ für alle N Wörter unserer Liste gilt,
7. und semantisch ähnliche Wörter w_1, w_2 nach Kodieren in $\mathbb{R}^K$ einen kleinen Abstand $d(e(w_1), e(w_2)) \approx 0$ haben.

Punkt 6 allein wäre ein gewöhnlicher Autoencoder auf One-Hot-Vektoren – und der lernt nichts über Bedeutung, denn One-Hot-Vektoren sind alle gleich weit voneinander entfernt. Punkt 7 ist der Clou.

Kernidee

Die Semantik eines Wortes ergibt sich durch seine Nutzung in Sätzen (*distributional hypothesis*). Beispiel: „Die [MASK] frisst ihr Futter am liebsten aus dem goldenen Napf“ kommt in Texten mit [MASK] = Katze und [MASK] = Hündin vor. Der Machine-Learning-Task, unsupervised: Gegeben Text, ersetze ein Wort durch [MASK] und lasse vorhersagen, was [MASK] ist – wie ein Denoising Autoencoder, bei dem das Rauschen im Löschen eines Wortes besteht.

CBOW (Continuous Bag of Words, für kleinere Korpora). Hyperparameter ist die Fenstergröße S (window size), z.B. $S = 2$. Eingabe: $2S + 1$ Token, die hintereinander im Text vorkommen, dabei das mittlere durch [MASK] ersetzt – also $2S$ One-Hot-Vektoren. Ausgabe: eine Wahrscheinlichkeitsverteilung auf allen N Wörtern. Loss-Funktion: negative Log-Likelihood (NLL), d.h. Vergleich der ausgegebenen Verteilung mit dem One-Hot-Vektor für das Wort, welches sich hinter [MASK] verbirgt. Architektur: 1 Hidden Layer mit K Neuronen, der „Encoder“ $W \in \mathbb{R}^{K \times N}$, angewendet auf alle $2S$ Kontextwörter und dann gemittelt (*averaging*), danach ein Layer „Decoder“ $W' \in \mathbb{R}^{N \times K}$ gefolgt von Softmax:

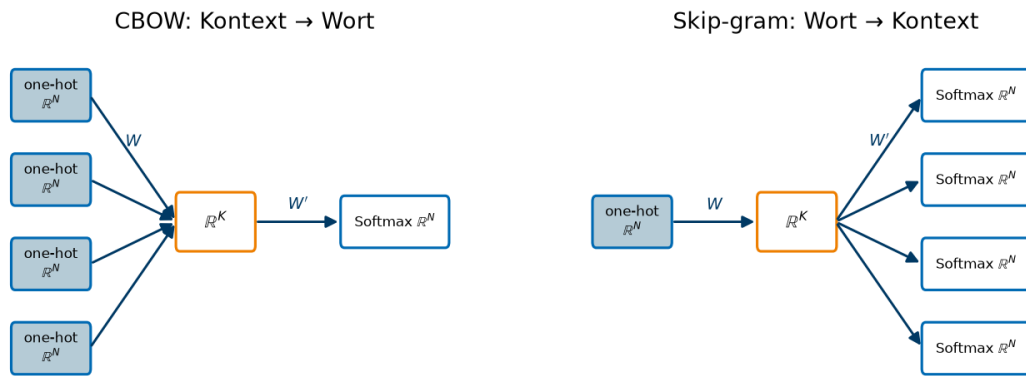
$$P(w_{\text{mitte}} | \text{Kontext}) = \text{sm}\left(W' \cdot \frac{1}{2S} \sum_{j \in \text{Kontext}} W w_j\right). \quad (123)$$

Die Wortvektoren sind die Spalten von W : $e(w) = Ww$ ist für einen One-Hot-Vektor w einfach die entsprechende Spalte.

Skip-gram, die Alternative zu CBOW: Gegeben ein Token, sage den Kontext vorher. Eingabe: das mittlere Token als One-Hot-Vektor für eine Abfolge von $2S + 1$ Token. Ausgabe: eine Wahrscheinlichkeitsverteilung auf allen N Token. Loss-Funktion: NLL, d.h. Vergleich der ausgegebenen Verteilung mit der Summe der One-Hot-Vektoren für alle $2S$ Kontexttoken. Architektur: 1 Hidden Layer mit K Neuronen (der „Encoder“), danach ein Layer „Decoder“ mit N Neuronen gefolgt von Softmax. Skip-gram funktioniert bei großen Korpora und seltenen Wörtern besser.

word2vec benutzen

Verfügbare Gewichte W gibt es mit Gensim (z.B. **wiki-english-20171001**) oder von Hugging Face (z.B. **word2vec-google-news-300**, $N = 3$ Millionen, $K = 300$). Klappt nicht gut? Drop-in-Ersatz für word2vec ist Facebooks **fastText**, welches hauptsächlich einen geeigneteren Tokenizer benutzt (Stichwort Subword-Tokenization) und dadurch auch mit unbekannten Wörtern umgehen kann.



CBOW und Skip-gram: dieselben zwei Matrizen W (Encoder, $N \rightarrow K$) und W' (Decoder, $K \rightarrow N$), einmal von mehreren Kontextwörtern auf ein Wort, einmal umgekehrt.

word2vec selbst trainieren (ein ganz knapper Überblick)

Zutatenliste: Keine GPU oder andere Beschleuniger zur Hand? Google Colab oder CPU reicht. Input: ein Textkorpus zum Training, eine Tokenization-Strategie bzw. ein Tokenizer (darüber sprechen wir noch), die Dimension K des Latent Space, CBOW oder Skip-gram (wie groß ist das Korpus?), von Grund auf trainieren (Pretraining) oder Finetuning, und welche Implementierung?

- **Gensim** ([Tutorial](#)). Pro: NLP-Gesamtlösung, gut dokumentiertes Paket. Contra: keine GPU-Beschleunigung (aber für word2vec ist das „egal“), und keine Kombination von word2vec im Training mit anderen Tasks (wie wir es beim Denoising Label Autoencoder getan haben).
- **PyTorch**, „so richtig von Hand“: [Skip-gram implementation step by step](#); oder mit dem `torch.nn.Embedding`-Layer (das ist die Matrix W , allerdings **transponiert** gespeichert: `nn.Embedding(N, K).weight` hat Shape (N, K) , eine *Zeile* pro Wort, und die Multiplikation mit dem One-Hot-Vektor ist als Lookup implementiert): [Word Embeddings: Encoding Lexical Semantics](#), [d2l.ai](#).

Wozu? Durch Pretraining oder Finetuning von word2vec-Modellen mit PyTorch bekommt man Wortvektoren, die auf die eigene Domäne (Fachsprache, andere Sprache) passen, und kann sie als erste Schicht in ein größeres Modell einbauen.

4.3.2 Von Wörtern zu Dokumenten: NLP

Beispiel-Task: ähnliche Dokumente finden

Gegeben ist eine Menge von N (kurzen) Dokumenten $\{D_1, \dots, D_N\}$ (z.B. Tweets, Webseiten, Hausarbeiten). Gesucht sind k Paare ähnlicher Dokumente ($k \ll N$), z.B. um Tweets thematisch zu gruppieren oder Plagiate zu finden: $\{(x_1, y_1), \dots, (x_k, y_k)\}$, Paare von Indizes, sodass D_{x_i} und D_{y_i} ähnlich sind.

Klassischer Ansatz: Wörter in D_i zählen, Häufigkeitsverteilungen von D_i und D_j vergleichen (*bag of words*). Probleme mit diesem Ansatz:

1. „Katze isst Maus“ und „Maus isst Katze“ sind ununterscheidbar.
2. „Schwimmen“ und „schwamm“ haben nichts miteinander zu tun.
3. „Hund“ und „Tier“ haben nichts miteinander zu tun.
4. Wörter wie „der“, „die“, „das“ sind sehr häufig, aber bedeutungslos.

Klassische Lösungsansätze: (1) n -Gramme statt Wörter betrachten, (2) Lemmatisierung, (3) komplexe Ontologien (formalisiertes Wissen), (4) Stopwords entfernen.

Wörter? Token!

Alternative Lösung zu Problem 2 (statt Lemmatisierung): Buchstaben bzw. Zeichen anstelle von Wörtern als „Atome“ (= Token) nehmen, und n -Gramme davon (z.B. Trigramme, $n = 3$). Beispiel: „schwimm!“ und „schwimmen“ ergeben

$$[\text{sch, chw, hwi, wim, imm, mm!}] \quad \text{und} \quad [\text{sch, chw, hwi, wim, imm, mme, men}], \quad (124)$$

die haben sehr ähnliche Trigrammhäufigkeitsverteilungen. Konkret zu klärende Fragen: Leerzeichen wie jedes andere Zeichen behandeln? Über das Satzende „.“ hinaus? Flexibles n , z.B. Silben statt Buchstaben-trigramme? Die Antworten darauf bilden die **Tokenisierungsstrategie**, den Tokenizer. Der Einfachheit halber bleiben wir zunächst bei Token = Wörter, d.h. wir löschen Satzzeichen und trennen an Leerzeichen; Byte Pair Encoding, der Tokenizer moderner LLMs, kommt im [Kapitel „Transformer“](#).

Modernerer Ansatz (supervised): mit einem neuronalen Netz; Menschen labeln Ähnlichkeit. Eingabe: Paare (D_i, D_j) , Ausgabe: Ähnlichkeitsschätzung. **Unsupervised mit neuronalen Netzen**:

1. Alle Dokumente D_i einlesen als Liste von Wörtern.
2. Jedes einzelne Wort mit einem zuvor trainierten neuronalen Netz (word2vec) umrechnen in eine andere Repräsentation, die sich besser vergleichen lässt.
3. Für jedes Paar (D_i, D_j) die Listen von Repräsentationen vergleichen (z.B. Kosinus-Ähnlichkeit der Mittelwerte), daraus ein Ähnlichkeitsmaß für das Paar berechnen.
4. Die Top- k ähnlichsten Paare ausgeben.

Problem 3 („Hund“ und „Tier“) löst word2vec, weil beide in ähnlichen Kontexten vorkommen und daher nahe beieinander liegen. Problem 1 (Wortreihenfolge) löst es nicht – dafür braucht man Modelle, die Sequenzen verarbeiten ([Kapitel „Recurrent Neural Networks“](#) und [„Transformer“](#)).

Literatur und Softwarepakete

- Das modernste Lehrbuch ist Dan Jurafsky, James H. Martin: *Speech and Language Processing*. Es gibt einen „Third Edition Draft“ frei verfügbar: web.stanford.edu/~jurafsky/slp3/.
- Das Python Natural Language Toolkit (NLTK), nltk.org/book, enthält z.B. den PunktTokenizer, der so arbeitet wie wir jetzt.
- Radim Řehůřek (& Koautoren) gensim-Bibliothek für Python, radimrehurek.com/gensim.

4.3.3 Übung

Der Satz „der hund bellt laut im garten“ (6 Token), Fenstergröße $S = 2$.

1. Wie viele (Kontext, Zielwort)-Paare erzeugt Skip-gram daraus? Schreiben Sie die Paare für das Zielwort „bellt“ auf. *Tipp: Am Satzanfang und -ende ist das Fenster kleiner.*
2. Was ist bei CBOW Eingabe und was Ausgabe – und wie herum ist es bei Skip-gram?
3. Der Embedding-Layer für ein Vokabular von 50 000 Wörtern und 300 Dimensionen: Wie viele Parameter, und was steht in Zeile i ?

Antwortprobe: Bei (1) liefern das erste und das letzte Token je zwei Paare, das dritte und das vierte je vier; bei (3) sind es 15 Millionen.

4.4 Representation Learning und Graphen

4.4.1 Representation Learning

Worum geht's?

Gegeben Daten, die einer unbekannten Verteilung folgen, wollen wir *automatisch* Features entwickeln, die spätere Aufgaben mit diesen Daten erleichtern. Spätere Aufgaben können sein: die Verteilung schätzen (um daraus zu sampeln, z.B. Text generieren), Klassifikation (z.B. Spam/Ham), Ähnlichkeit bestimmen (z.B. zwischen zwei Texten).

Oberste Ziele sind eine höhere Modellperformance für die späteren Aufgaben und bessere Generalisierbarkeit. Ein weiteres Ziel ist möglicher **Transfer**: Features, die für eine Aufgabe gelernt wurden, für eine andere nutzen (Beispiel: Übersetzungen – ein Encoder für Englisch, der auch für Französisch etwas taugt).

Wie genau? Zwei Mechanismen haben wir bereits gesehen: Dimensionsreduktion durch „intelligentes Downsampling“ bzw. Kompression (wie beim Autoencoder), und nichtlineare neue Features aus alten Features entwickeln (jede versteckte Schicht tut das).

Andere Features, die man manuell hinzufügen kann

Representation Learning ist die automatische Variante von *Feature Engineering*. Zwei klassische, handgemachte Beispiele:

- Das **Spektrogramm** einer Audiodatei. So wird eine Waveform-Datei (z.B. eine Unterhaltung, ein Song) in ein Bild umgewandelt, indem die (Kurzzeit-)Fourier-Transformation genutzt wird. Relevant für Sprachsynthese und Spracherkennung.
- Enthält ein Bild horizontale Kanten? Vertikale Kanten? Ecken? Ein Klassifikator für „das Bild enthält ein Haus“ ist leichter zu trainieren, wenn nicht nur Pixelwerte eingehen, sondern auch die Positionen von Ecken und Kanten. Die automatische Variante davon sind **CNNs** (Convolutional Neural Networks), siehe das [Kapitel „Faltung und CNNs“](#).

4.4.2 node2vec

word2vec aus anderer Perspektive

Ein Satz, wie er bei word2vec eingeht, kann als linearer Graph modelliert werden: „Die → Katze → frisst → Mäuse“. Mehrere Sätze können gleiche Wörter enthalten, damit bilden mehrere Sätze größere Graphen mit linearen Teilgraphen: „Der → Hund → frisst → Leckerli“ teilt sich den Knoten „frisst“ mit dem ersten Satz.

Mit diesem Graphen allein lassen sich die Sätze nicht zurückgewinnen, denn „Der Hund frisst Mäuse“ sowie „Die Katze frisst Leckerli“ sind keine ursprünglichen Sätze, aber Pfade im Graphen. Besser wäre ein probabilistisches Modell auf Grundlage des Graphen: für jeden Knoten x eine Wahrscheinlichkeitsverteilung auf den Knoten y mit Kante $x \rightarrow y$, abhängig von der vorherigen Kante $z \rightarrow x$ (eine Markovkette zweiter Ordnung).

Aus anderer Perspektive: von Graphen zu Sätzen

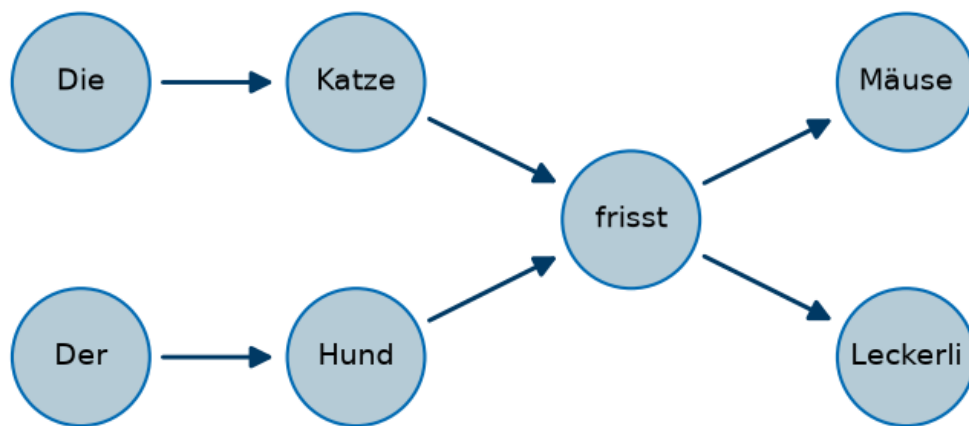
Umgekehrt: Gegeben ein Graph, lassen sich daraus „Sätze“ bilden durch wiederholtes Verfolgen einer ausgehenden Kante. Vereinfachtes probabilistisches Modell auf Grundlage des Graphen: für jeden Knoten x eine Gleichverteilung auf den Knoten y mit Kante $x \rightarrow y$, unabhängig von der vorherigen Kante. Das ist ein **Random Walk**.

Beispiel mit einem Graphen aus der Wikipedia: Von „Katze“ führen Links zu „Maus“, „Mäuse“, „Sendung mit der Maus“; von „Maus“ zu „Computermouse“, „Lesk-Algorithmus“ (Wortsinn-Disambiguierung), ...Daraus lassen sich endliche Random Walks bilden. Ziel: Verstehe einen Knoten durch seinen Kontext. Beispiel: Haushund und Hausmaus haben gemeinsame eingehende Links.

node2vec praktisch

Bei **node2vec** (Grover & Leskovec, 2016) haben wir zwei Schritte:

1. Generiere „Sätze“ (Knoten, [Kontext]) durch Random Walks auf dem Graphen.
2. Benutze word2vec-Skip-gram-Training auf den Sätzen.



Zwei Sätze als Graph: Der gemeinsame Knoten „frisst“ erzeugt Pfade, die keine ursprünglichen Sätze sind.

Schritt 1 ist sehr ressourcenintensiv, kann aber parallelisiert werden. Software dafür: [StellarGraph](#) (OK API, nutzt gensim), eine schnelle Implementierung mit Apache Spark ([arXiv:1805.00280](#)), und als Modul für PyTorch [torch_geometric.nn.models.Node2Vec](#) (siehe auch das [Beispiel](#)).

Beispielanwendungen: [Recommendations using node2vec](#) – aus einem Netflix-Datensatz werden Serien und Filme sowie deren Genres extrahiert und zu einem bipartiten Graphen Titel → Genre gebaut; nach node2vec kann man zu einem Titel im Latent Space nach ähnlichen Titeln suchen. Ähnlich, mit mehr Klassen von Knoten: [Knowledge graph analysis with node2vec](#).

Denkaufgabe

Wie könnte *wav2vec* aussehen – word2vec für Audio? Was sind die „Wörter“, was ist der „Kontext“?

Graph Neural Networks

Etwas anderes: Anstatt die Struktur des Graphen in (Knoten, Kontext)-Paare zu entwickeln, lässt sich die Struktur des Graphen auch direkt im neuronalen Netz nutzen. Beispiel: Ein Graph mit Knoten Kaffee, Tee, Kanne, Tasse, Mühle. Jeder Knoten trägt einen Feature-Vektor, und in jeder Schicht des Netzes aktualisiert jeder Knoten seinen Vektor aus den Vektoren seiner Nachbarn – **Message Passing**. Mit der Adjazenzmatrix A des Graphen ist das eine Matrixmultiplikation: Stehen die Knotenvektoren als Zeilen in H , so enthält AH für jeden Knoten die Summe der Vektoren seiner Nachbarn. Die Gewichte sind für alle Knoten gleich, sodass das Netz auf Graphen beliebiger Größe funktioniert. Das ist der Grundbaustein von Graph Neural Networks; wir vertiefen das hier nicht.

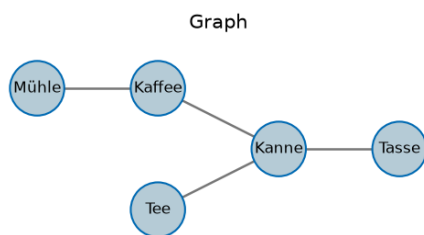
Vertiefung: die Message-Passing-Formel

$$h_v^{(k+1)} = \tau \left(W^{(k)} h_v^{(k)} + \sum_{u \in N(v)} U^{(k)} h_u^{(k)} \right). \quad (125)$$

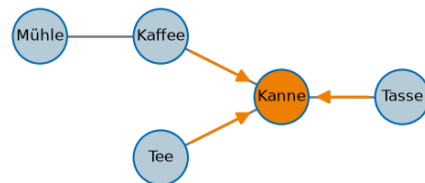
Dabei ist $N(v)$ die Nachbarschaft von v ; $W^{(k)}$ verarbeitet den eigenen Vektor, $U^{(k)}$ die Nachrichten der Nachbarn. In Matrixform, mit den Knotenvektoren als Zeilen von $H^{(k)}$: $H^{(k+1)} = \tau(H^{(k)}W^{(k)T} + AH^{(k)}U^{(k)T})$. Varianten unterscheiden sich darin, wie die Nachbarn gewichtet werden (Mittelwert statt Summe, Normierung mit dem Grad, gelernte Attention).

Weiterlesen

Prince, *UDL*, Kap. 13 (Graph neural networks). Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).



Message Passing: Kanne sammelt von Nachbarn



Message Passing in einem GNN: Jeder Knoten aktualisiert seinen Vektor aus den Vektoren seiner Nachbarn, hier für „Kanne“.

4.5 Recurrent Neural Networks

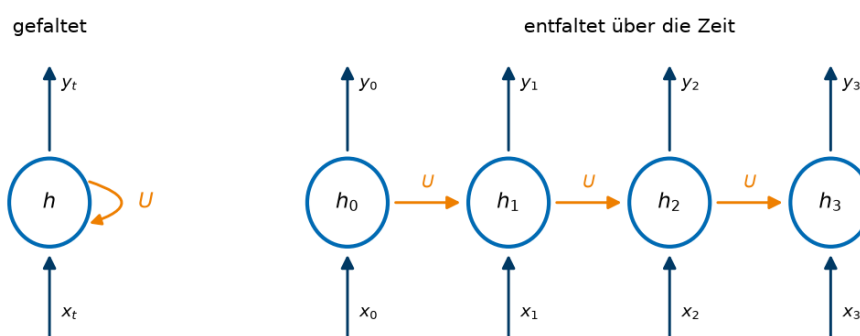
4.5.1 Recurrent Neural Networks

RNNs

Input: ein linearer Strom von Daten, z.B. Text, Audio, Video oder andere Zeitreihen. Output: irgendeine vektorielle oder skalare Größe, z.B. die Vorhersage des nächsten Wortes. Idee: Zum Zeitpunkt t nicht nur die neuen Daten x_t , sondern auch die Ergebnisse aus den Daten von $t - 1$ verwenden, um zum Ergebnis zu kommen. Ein rekurrentes Neuron hat einen **hidden state** h_t :

$$h_t = \tau(Wx_t + Uh_{t-1} + b), \quad y_t = Vh_t. \quad (126)$$

In der „entfalteten Perspektive“ (unrolled) ist ein RNN über T Zeitschritte ein tiefes Netz mit T Schichten, die sich alle dieselben Gewichte W, U, V teilen. Backpropagation funktioniert dann ganz normal (*backpropagation through time*) – mit dem Problem, dass Gradienten über viele Zeitschritte hinweg exponentiell verschwinden oder explodieren, weil immer wieder mit U multipliziert wird. Die Lösung dafür sind LSTMs.



Ein rekurrentes Neuron: gefaltet mit der Rückkopplung U (links) und über vier Zeitschritte entfaltet (rechts) – ein tiefes Netz mit geteilten Gewichten.

Historie: Rekurrente Netze

1982 Hopfield-Netzwerk · 1985 Boltzmann Machine (Hinton, Sejnowski) · 1980er RNNs · 1991 Schmidhubers frühe Attention · 1990er Gated RNNs · 1997 LSTM (Hochreiter, Schmidhuber) · 2014 GRUs (Cho, Bengio et al.) · 2017 Transformer (Vaswani et al.).

Im Gegensatz zur *Finite Impulse Response* (FIR) eines CNNs (machen wir noch) ist ein RNN ein *Infinite Impulse Response* (IIR) Signalverarbeiter: Ein Eingangssignal zum Zeitpunkt t kann den Zustand prinzipiell für immer beeinflussen.

Trick: Skip Connections

Auch bekannt als *Residual Connections*. Anstatt $x^{(m)} = f_{\vartheta}(x^{(m-1)})$ (z.B. $f = \text{MLP}$) parametrisiere

$$x^{(m)} = x^{(m-1)} + \text{res}(x^{(m-1)}), \quad \text{res}(x^{(m-1)}) = f_{\vartheta}(x^{(m-1)}). \quad (127)$$

Idee: Jeder Layer ist nah dran an „nichts ändern“ (Identität), und der Gradient hat immer einen direkten Weg (den Term $x^{(m-1)}$) an der Schicht vorbei. Hilft gegen verschwindende oder explodierende Gradienten – in RNNs, in ResNets (daher der Name) und in Transformern.

Gated RNNs

GRU (Gated Recurrent Unit): Der hidden state ist gleichzeitig der Output. Aus Input und altem hidden state wird ein *Kandidat* für den neuen hidden state berechnet, und zwei Gates (Sigmoides, also Werte in $(0, 1)$) steuern: Das *Reset Gate*, wie viel vom alten Zustand in den Kandidaten eingeht, und das *Update Gate*, wie stark der Zustand zum Kandidaten hin aktualisiert wird.

LSTM (Long Short-Term Memory): Zusätzlich zum hidden state h_t (= Output, *short term memory*) gibt es einen **cell state** c_t (*long term memory*), der nur durch Addition und komponentenweise Multiplikation verändert wird – eine Art Skip Connection durch die Zeit. Drei Gates (Sigmoides) steuern ihn:

forget (was aus c_{t-1} gelöscht wird), *input* (was neu hineingeschrieben wird), *output* (was davon in h_t sichtbar wird). Mehr Erklärungen: Shi Yan @ MLReview, *Understanding LSTM and its diagrams*.

Schaltbilder der Zellen: [LSTM](#) und [GRU](#) auf Wikipedia (Abbildungen CC-BY-SA); die Gates sind dort als σ -Knoten gezeichnet, die elementweise multiplizieren, also genau die stetigen Bitmasken aus dem [Kapitel „Netze von Hand bauen“](#).

Bill Gates (2018): „huge milestone in advancing artificial intelligence“ – weil OpenAI mit LSTMs Dota 2 gewann. Auch mit LSTMs gebaut: AlphaStar von Google DeepMind, spielt StarCraft 2 (2019).

LSTM praktisch

Mit PyTorch:

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

```
class MyModel(nn.Module):
    def __init__(self, embedding_length, hidden_size, output_size):
        super().__init__()
        self.lstm = nn.LSTM(embedding_length, hidden_size)
        self.label = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h, (h_n, c_n) = self.lstm(x)
        return self.label(h)
```

`nn.LSTM` bekommt eine Sequenz von Vektoren (z.B. word2vec-Embeddings der Wörter) und liefert die Sequenz der hidden states; darauf ein linearer Layer für die Vorhersage. **Achtung bei den Achsen:** Anders als sonst in PyTorch erwartet `nn.LSTM` voreingestellt (`seq, batch, feature`), der Batch liegt also auf der *zweiten* Achse (siehe [Kapitel „Tensoren“](#)). Mit `batch_first=True` im Konstruktor gilt wieder die gewohnte Reihenfolge.

Am Rande: Konkatenieren vs. Summe

Wenn wir zwei verschiedene Eingabevektoren in einer Schicht verarbeiten wollen, ist es naheliegend, diese zu einem längeren Vektor zu konkatenieren. Oft sehen wir aber im Code stattdessen ein einfaches $+$. Wie kann das funktionieren???

Eine einfachere Frage: Unter welchen Umständen lässt sich für zwei Vektoren $v, w \in \mathbb{R}^n$ aus der Summe $v + w \in \mathbb{R}^n$ wieder das Paar (v, w) rekonstruieren? Probieren Sie explizite Beispiele mit $n = 2$ aus, etwa $v = (3, 0)$ und $w = (1, 1)$. Überlegen Sie hinterher, wie das bei der eigentlichen Frage weiterhilft. (Stichwort: Wenn v und w aus komplementären Unterräumen kommen – oder das Netz lernen kann, sie dort hinzulegen –, verliert die Summe nichts.)

4.5.2 Übung

Ein RNN mit $h_t = \tanh(w x_t + u h_{t-1})$, $w = 1$, $u = 0,5$, $h_0 = 0$ und der Eingabefolge $x = (1, 0, 0)$.

1. Berechnen Sie h_1, h_2, h_3 (zwei Nachkommastellen genügen).
2. Wiederholen Sie mit $u = 0,1$ und mit $u = 0,9$. Wie lange „erinnert“ sich das Netz jeweils an die 1 vom Anfang? *Tipp: Nach dem ersten Schritt ist die Eingabe 0, es bleibt also eine Multiplikation mit u pro Schritt.*

Antwortprobe: Bei (2) fällt einer der drei Verläufe deutlich langsamer ab als die anderen.

4.6 Transformer

4.6.1 Transformer

Historie: BLEU (2001)

Um Übersetzungsmodelle zu vergleichen, braucht man ein Maß. Papineni et al. (2001): „Human evaluations of machine translation are extensive but expensive. [...] We propose a method of automatic machine translation evaluation that is quick, inexpensive, and language-independent, that correlates highly with human evaluation.“ Die Grundidee: „The closer a machine translation is to a professional human translation, the better it is.“

Der **BLEU-Score** (bilingual evaluation understudy) zwischen 0 und 1 (auch skaliert 0–10 oder 0–100) ist das geometrische Mittel aus den n -Gramm-Präzisionsscores der Maschinenübersetzung relativ zu einer menschlichen Referenzübersetzung, multipliziert mit einem Strafterm für abweichende Übersetzungslänge (*brevity penalty*). Die n -Gramm-Präzision ist die Anzahl (mit Mehrfachzählung, aber gedeckelt durch die Referenz) der n -Gramme, die in der Maschinenübersetzung vorkommen und in der Referenzübersetzung, normiert mit der Anzahl der n -Gramme der Maschinenübersetzung. Für BLEU ist $n = 1, 2, 3, 4$ üblich.

Historie: NeurIPS 2017

Vaswani et al., *Attention is all you need*: „The dominant sequence transduction models are based on complex recurrent or convolutional neural networks in an encoder-decoder configuration. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. [...] Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles by over 2 BLEU. [...] a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs.“

Zur Einordnung: WMT = Workshop on Statistical Machine Translation, Task und Datensatz; die Challenge läuft bis heute jährlich weiter; sacreBLEU als Standard-Implementierung; 41.8 „BLEU“ = 0,418 BLEU. Und das WMT19-Modell aus dem [Kapitel „Vortrainierte Modelle“](#) ist genau ein Nachfolger davon.

Transformer technisch

Der Aufbau (Fig. 1 in „Attention is all you need“): ein **Encoder**-Stapel und ein **Decoder**-Stapel, beide aus identischen Blöcken mit Multi-Head Self-Attention, einem MLP, Residual Connections und Layer-Normalisierung. Anstelle von Rekurrenz bekommt jedes Token eine **Positionskodierung** addiert. Der Kern ist die **Attention**:

$$\text{Attention}(Q, K, V) = \text{sm}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (128)$$

wobei Queries Q , Keys K und Values V lineare Projektionen der Token-Vektoren sind: Jedes Token „fragt“ mit seiner Query alle anderen Tokens, vergleicht sie mit deren Keys (Skalarprodukt, Softmax $\rightarrow$ Gewichte) und sammelt eine gewichtete Summe ihrer Values ein. Das ist eine Schicht, in der jedes Token von jedem anderen abhängt – ohne Reihenfolge, daher die Positionskodierung – und die vollständig parallel berechnet werden kann. Wir implementieren keine Transformer in dieser Vorlesung; im Kapitel [„LLMs benutzen“](#) benutzen wir sie.

Das Architekturbild ist Figure 1 in Vaswani et al., *Attention Is All You Need* ([arXiv:1706.03762](#)): links der Encoder-Stapel, rechts der Decoder-Stapel, dazwischen die Cross-Attention. Eine annotierte, ausführbare Fassung ist [The Annotated Transformer](#).

Konkrete Modelle mit Transformern

GPT (Generative Pretrained Transformer): Trainiert unsupervised auf gigantischen Textmassen (präziser: self-supervised); BPE (Byte Pair Encoding) als Tokenization-Schema; sehr große Modelle (= sehr viele Layer, sehr viele Parameter); gelernte Positionskodierung, begrenzter Kontext; **Decoder-only**: das nächste Token vorhersagen. Varianten: GPT-2, GPT-J, GPT-NeoX, Llama 1/2/3, und nichtöffentlich GPT-3, 3.5, 4, etc.

Byte Pair Encoding (BPE): ein Tokenization-Schema (Gage 1994). BPE baut auf einem Tokenizer auf, meist Token = Byte oder Character. Algorithmus:

1. Zähle, wie oft jedes benachbarte Tokenpaar im Korpus vorkommt.

2. Nimm das **häufigste Paar** und ersetze jedes Vorkommen durch ein einzelnes neues Token; das Vokabular wächst um eins.
3. Zurück zu Schritt 1 – bis es genug verschiedene Token gibt.

Die Vokabulargröße bei GPT-4s BPE „tiktoken“ ist z.B. 100 256; durchschnittlich entspricht ein Token etwa 4 Bytes. Häufige Wörter werden zu einem Token, seltene in Silben oder Buchstaben zerlegt – das ist die „flexible n “-Antwort auf die Tokenisierungsfrage aus dem letzten Kapitel.

NanoGPT: Visualisierung und Erklärung, Layer für Layer, von Brendan Bycroft: bbycroft.net/llm.

BERT (Bidirectional Encoder Representations from Transformers): **Encoder-only** (nicht zwangsläufig generativ); Tokenization auf Subwort-Level (WordPiece); Training mit 2 Tasks: Masked Language Modeling (ähnlich wie bei word2vec, aber mit Kontext in beide Richtungen) und „nächsten Satz vorhersagen“. Bei gleicher Modellgröße besser als GPT für manche Tasks (Klassifikation, Embeddings). Varianten: RoBERTa, ALBERT, DistilBERT.

ViT (Vision Transformer, Dosovitskiy et al., „An Image is Worth 16x16 Words“): Das Bild wird grob gerastert in „Patches“ aufgeteilt; jeder Patch (= Token) wird in einen Vektor umgewandelt; Positionskodierung für 2D-Position statt 1D wie bei Text; ansonsten „klassischer“ Transformer. Für Klassifikationsaufgaben manchmal besser als CNNs.

Implementierung des Transformers

Alexander Rush et al.: *The Annotated Transformer* (updated 2022 edition) – der Artikel „Attention is all you need“ zusammen mit Codeschnipseln und einigen Kommentaren. Für Verrückte oder sehr Interessierte: „I made a transformer by hand (no training!)“ von Theia Vogel, mit sorgfältig von Hand bestimmten Gewichten – die Fortsetzung unseres „Parameter von Hand wählen“ aus dem [Kapitel „Netze von Hand bauen“](#).

4.6.2 Übung

Drei Token mit Query-Key-Skalarprodukten $q \cdot k_1 = 2$, $q \cdot k_2 = 2$, $q \cdot k_3 = 0$ und Values $v_1 = 1$, $v_2 = 3$, $v_3 = 10$.

1. Berechnen Sie die Attention-Gewichte mit Softmax (ohne Skalierung) und daraus die Ausgabe.
Tipp: Zwei der drei Exponenten sind gleich; $e^2 \approx 7,39$.
2. Die Gewichte müssen sich zu 1 summieren – prüfen Sie das nach.
3. Warum braucht ein Transformer Positionsempeddings, ein RNN aber nicht?

Antwortprobe: Das dritte Gewicht ist kleiner als $\frac{1}{10}$, aber nicht null, und die Ausgabe bleibt trotz $v_3 = 10$ unter 2,6.

Weiterlesen

Prince, *UDL*, Kap. 12 (Transformers). Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).

5 Architekturen und Anwendungen

5.1 Faltung und Convolutional Neural Networks

5.1.1 Motivation zur Faltung

Bildverarbeitung: it's a blur

Wer schon einmal ein Bild in GIMP oder Photoshop weichgezeichnet hat, hat eine Faltung benutzt: Jeder Pixel wird durch einen gewichteten Mittelwert seiner Nachbarn ersetzt. Und wer Kanten gefunden hat („make a difference“): Das Originalbild minus das weichgezeichnete Bild lässt genau die Stellen übrig, an denen sich benachbarte Pixel stark unterscheiden – die Kanten.



Faltung in der Bildverarbeitung: Original, mit einer Gaußfunktion weichgezeichnet, und die Differenz beider – die Kanten.

Das menschliche Auge erkennt Kanten(orientierung)

Hubel & Wiesel (1959) haben in der Sehrinde von Katzen Neuronen gefunden, die auf Balken einer bestimmten *Orientierung* antworten – und auf andere Orientierungen nicht (Reproduktion von Wurtz 1969, in Robert H. Wurtz: „Recounting the impact of Hubel and Wiesel“, J Physiol 2009). Solche Neuronen sind lokale Kantendetektoren: Sie schauen auf einen kleinen Ausschnitt des Gesichtsfelds und reagieren auf ein bestimmtes Muster darin. CNNs sind die technische Umsetzung dieser Idee, und die ersten Schichten eines trainierten CNN sehen tatsächlich aus wie Kantendetektoren verschiedener Orientierung.

Das klassische Bild dazu, ein Neuron, das nur bei einer Balkenorientierung feuert, steht im Wikipedia-Artikel zu [Hubel und Wiesel](#) (Orientation column, Abbildung CC-BY-SA).

In der Informatik

Abgesehen von CNNs ist die Faltung das zentrale Instrument in der Signalverarbeitung. Das Eindimensionale ist leichter zu verstehen als das Zweidimensionale, damit fangen wir an.

5.1.2 Eindimensionale Faltung

Definition 5.1 (Diskrete Faltung). Wenn $f: [-n, n] \rightarrow \mathbb{R}$ und $g: [0, N] \rightarrow \mathbb{R}$ (auf den ganzen Zahlen), dann ist $f * g: [0, N] \rightarrow \mathbb{R}$ definiert durch

$$(f * g)(x) = \sum_{y=-n}^n f(y) g(x - y), \quad (129)$$

wobei der Definitionsbereich von g mit 0 erweitert wird (zero padding).

Eine Interpretation: g ist das Signal, f der sogenannte **Filter** (oder Kernel). Der Filter ist kurz ($2n+1$ Werte), das Signal lang.

Interpretation der Definition

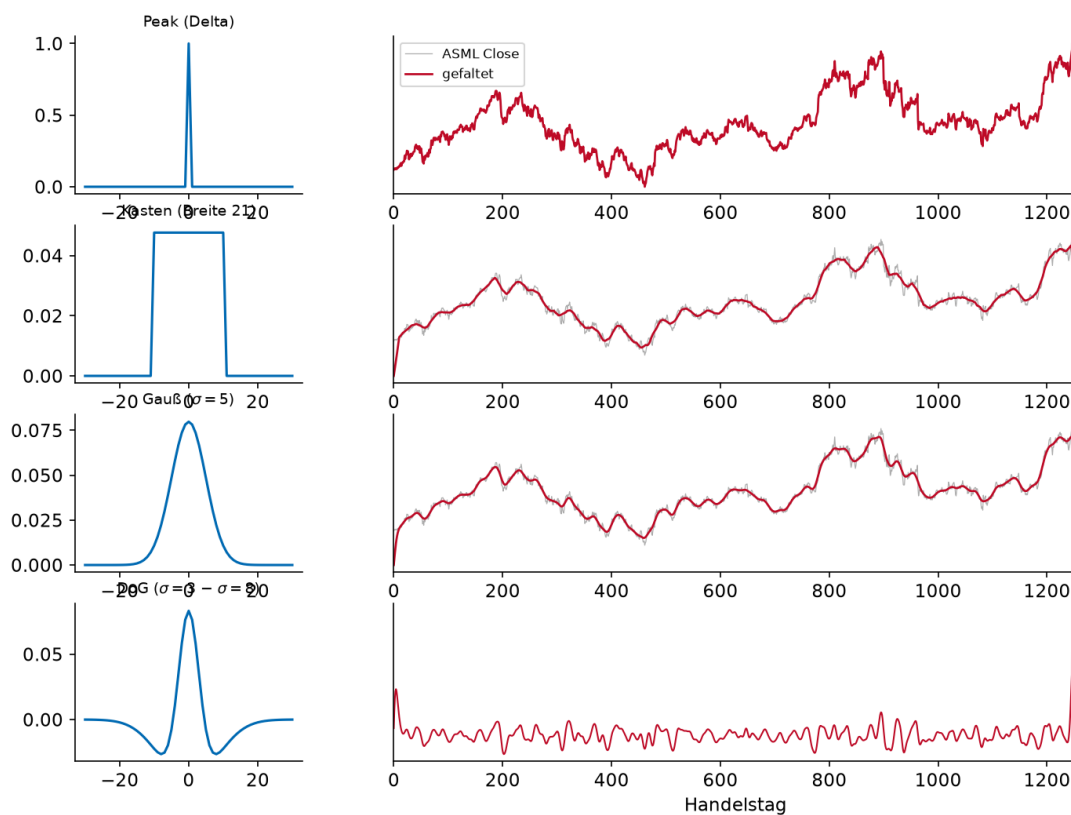
Da steht ein Skalarprodukt des Vektors $(f(-n), \dots, f(n))$ mit $(g(x+n), \dots, g(x-n))$. Rechnung: Nimm einen $[-n, +n]$ -Ausschnitt von g , zentriert bei x , dann *flippen* (bei x spiegeln) und dann Skalarprodukt mit f . Das Flippen ist der Unterschied zwischen Faltung und *Korrelation*; für symmetrische Filter ($f(-y) = f(y)$) ist es egal, und in Deep-Learning-Bibliotheken wird tatsächlich meist die Korrelation berechnet und trotzdem „Convolution“ genannt. Da die Filter gelernt werden, spielt es keine Rolle.

Ein Faltungsfilter ist also ein *Neuron mit $2n+1$ Gewichten*, das an jeder Position x des Signals dasselbe Skalarprodukt mit dem lokalen Ausschnitt berechnet. Dieselben Gewichte an jeder Position: das heißt **Parameter Sharing**, und es hat zwei Konsequenzen. Erstens hat die Schicht nur $2n+1$ Parameter, egal wie lang das Signal ist. Zweitens ist die Faltung **translationsäquvariant**: Verschiebt man das Signal, verschiebt sich das Ergebnis genauso.

Beispiel: ASML-Aktienkurs

Wir möchten ein Signal mit einem Filter falten. Das Signal ist der ASML-Aktienkurs in US-Dollar (closing price adjusted for splits, ein Teildatensatz über 5 Jahre); der Filter ist eine schmale Gaußfunktion (`gaussian_narrow`). Das Ergebnis ist der geglättete Kurs: Die Gaußfunktion ist ein gewichteter gleitender Mittelwert, und je breiter sie ist, desto glatter das Ergebnis. Faltung mit einem Peak (einer diskreten Delta-Funktion) gibt das Signal unverändert zurück; Faltung mit einer Kastenfunktion ist der ungewichtete gleitende Mittelwert; Faltung mit der Differenz zweier Gaußfunktionen (*Difference of Gaussians*, DoG) hebt Änderungen hervor und ist ein Kantendetektor. (Tipp: „DoG filter“ ist kein Snapchat-Filter.)

ASML-Aktienkurs (2020–2025) gefaltet mit verschiedenen Filtern



Der ASML-Aktienkurs (Schlusskurse 2020–2025) gefaltet mit vier Filtern. Delta lässt das Signal unverändert, Kasten und Gauß glätten, die Difference of Gaussians hebt Änderungen hervor.

Programmierung

Für NumPy-Arrays gibt es `scipy.signal.convolve`. Der Filter ist einfach ein Array von Funktionswerten auf einem Gitter, und `mode='same'` sorgt per Zero-Padding dafür, dass das Ergebnis so lang ist wie das Signal:

```
import numpy as np
import pandas as pd
from scipy.signal import convolve
```

```
data = pd.read_csv("ASML.csv")["Close"].to_numpy() # Schlusskurse als Array
```

```
x = np.linspace(-6, 6, 1000)
kernel = np.exp(-x**2 / (2 * 0.1**2)) # schmale Gaußfunktion
kernel /= kernel.sum() # normieren, sonst skaliert der Kurs
smoothed = convolve(data, kernel, mode='same') # data: die Schlusskurse als Array
```

Für Tensoren nimmt man `torch.nn.functional.conv1d`, und dort muss man sich an die Dimensionen gewöhnen: Eingabe und Filter haben den Shape `(batch, channels, length)` bzw. `(out_channels, in_channels, kernel_size)`, also für ein einzelnes Signal und einen einzelnen Filter zweimal `unsqueeze(0)`:

```
import torch
import torch.nn.functional as F

signal = torch.tensor(data, dtype=torch.float32).unsqueeze(0).unsqueeze(0) # (1, 1,
↪ N)
filt = torch.tensor([-1.0, -1, 1, 1, 1]).view(1, 1, 5) # (1, 1,
↪ 5)
out = F.conv1d(signal, filt, padding=2).squeeze() # wieder
↪ (N,)
```

Ein *Difference-of-Gaussians*-Filter entsteht als Differenz zweier normierter Gaußkerne verschiedener Breite:

```
def gaussian_1d(size, sigma):
    t = torch.arange(size).float() - size // 2
    g = torch.exp(-t**2 / (2 * sigma**2))
    return g / g.sum()

dog = gaussian_1d(11, sigma=1) - gaussian_1d(11, sigma=20)
out = F.conv1d(signal, dog.view(1, 1, -1), padding=5).squeeze()
```

Was man dabei beobachtet: Ein Faltungslayer, den man auf solchen Zeitreihen *lernen* lässt, findet von selbst Filter, die wie Gauß- oder DoG-Kerne aussehen – Glätten und Änderungen hervorheben sind schlicht die nützlichsten Operationen.

Das Faltungstheorem

Für zwei stetige Funktionen f, g und die stetige Faltung (Integral statt Summe) gilt für die Fourier-Transformation $\mathcal{F}$:

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g), \quad (130)$$

d.h. Faltung im Ortsraum ist Multiplikation im Frequenzraum. Auch umgekehrt: $\mathcal{F}(f \cdot g) = \mathcal{F}(f) * \mathcal{F}(g)$. Konkret: Mit einer reinen Sinuswelle zu falten heißt, den Koeffizienten dieser Welle in der Zerlegung des Signals herauszugreifen. Aus dem Faltungstheorem folgen auch die Assoziativität $(f * g) * h = f * (g * h)$ und die praktische Regel

$$\frac{d}{dx}(f * g) = f' * g = f * g', \quad (131)$$

d.h. „glätten und dann ableiten“ ist dasselbe wie „mit dem abgeleiteten Filter falten“ – man faltet also gleich mit der *Ableitung* der Gaußfunktion, statt erst zu glätten. Der DoG-Filter gehört eine Stufe höher: Die Differenz zweier Gaußfunktionen verschiedener Breite nähert die **zweite** Ableitung an (es ist $\partial G / \partial \sigma = \sigma \nabla^2 G$), und Kanten sind dort die Nulldurchgänge. Die erste Ableitung der Gaußfunktion ist eine *ungerade* Funktion, der DoG-Filter eine *gerade* – ähnlich sehen sie nur auf den ersten Blick aus. Das Faltungstheorem erklärt auch, warum die Gaußfunktion glättet: Ihre Fouriertransformierte ist wieder eine Gaußfunktion, die hohe Frequenzen (= schnelle Änderungen, Rauschen) dämpft. Kurzer Ausflug in die Fourier-Transformation? Andrew Zisserman, [Lecture 2](#).

5.1.3 Eindimensionales Pooling

Pooling $\approx$ Binning

Binning = Array in feste disjunkte Blöcke (ohne Überlapp) zerlegen, dann Summe oder Mittelwert bilden. Das ist ungefähr eine Faltung mit einer Kastenfunktion, bei der man aber nur wenige Werte des Ergebnisses verwendet. Wie viel Überlapp es gibt, bestimmt der **Stride** (Schrittlänge).

Pooling = Array in feste Blöcke (mit Überlapp) einteilen. **Average-Pooling** = Pooling + Mittelwert der Blöcke. **Max-Pooling** = Pooling + Maximum der Blöcke.

Pooling-Beispiel

Pooling eines Vektors $(a, b, c, d) \in \mathbb{R}^4$:

- Mit Stride 1 und Poolgröße 2: Pools $[(a, b), (b, c), (c, d)]$.
- Mit Stride 2 und Poolgröße 2: Pools $[(a, b), (c, d)]$; Average-Pooling gibt $[\text{mean}(a, b), \text{mean}(c, d)]$.
- Mit Stride 1 und Poolgröße 3: Pools $[(a, b, c), (b, c, d)]$; Max-Pooling gibt $[\max(a, b, c), \max(b, c, d)]$.

Von 4 Inputs reduzieren wir die Datenmenge also auf 3 oder 2 Outputs, je nach Größe und Stride. Pooling ist Dimensionsreduktion ohne Parameter, und Max-Pooling macht das Netz ein Stück weit *translationsinvariant*: Es ist egal, wo genau im Pool das Maximum lag.

Ein Stride größer als 1 kann man auch direkt bei der Faltung verwenden (*strided convolution*), dann reduziert die Faltung selbst die Dimension.

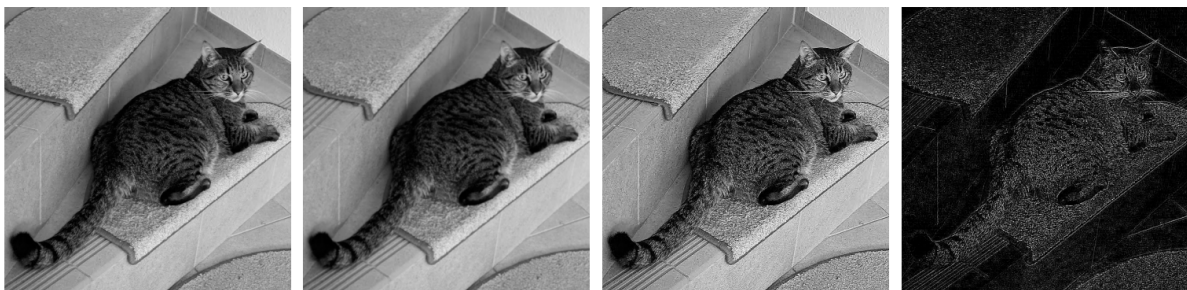
5.1.4 Zweidimensionale Faltung

Für Bilder $g: [0, N] \times [0, M] \rightarrow \mathbb{R}$ (Graustufen) und Filter $f: [-n, n]^2 \rightarrow \mathbb{R}$:

$$(f * g)(x_1, x_2) = \sum_{y_1=-n}^n \sum_{y_2=-n}^n f(y_1, y_2) g(x_1 - y_1, x_2 - y_2). \quad (132)$$

Ein 3×3 -Filter hat 9 Gewichte, die an jeder Pixelposition dasselbe Skalarprodukt mit der 3×3 -Nachbarschaft bilden. Beispiel mit GIMP: Unter *Filter* $\rightarrow$ *Allgemein* $\rightarrow$ *Faltungsmatrix* kann man die 9 Werte selbst eintragen und sieht unten den Output der 2D-Faltung – Weichzeichner, Schärfer, Kanten, Relief sind alles nur verschiedene Matrizen.

3×3-Faltungsmatrizen und ihr Ergebnis (wie in GIMP: Filter → Allgemein → Faltungsmatrix)											
Identität			Weichzeichnen			Schärfen			Kanten		
0	0	0	0.11	0.11	0.11	0	-1	0	-1	-1	-1
0	1	0	0.11	0.11	0.11	-1	5	-1	-1	8	-1
0	0	0	0.11	0.11	0.11	0	-1	0	-1	-1	-1



Vier 3×3 -Faltungsmatrizen und ihr Ergebnis auf demselben Graustufenbild ([notebooks/pallino.jpg](#)): Identität, Weichzeichnen (Mittelwert), Schärfer, Kanten. Genau so funktioniert der Dialog *Faltungsmatrix* in GIMP.

Bei Farbbildern hat die Eingabe 3 **Kanäle** (RGB), und ein Filter hat dann Shape $3 \times 3 \times 3$: Er summiert über alle Eingabekanäle. Eine Faltungsschicht mit C_{out} Filtern erzeugt C_{out} Ausgabekanäle (*Feature Maps*). Ein 1×1 -Filter schaut nur auf einen Pixel, aber über alle Kanäle – das ist ein linearer Layer pro Pixel und wird benutzt, um die Anzahl der Kanäle zu ändern, ohne die räumlichen Dimensionen anzurühren.

5.1.5 CNNs

Ein **Convolutional Neural Network** ist ein neuronales Netz, in dem (einige) Schichten Faltungen sind: `nn.Conv1d` bzw. `nn.Conv2d` mit lernbaren Filtergewichten, gefolgt von einer Aktivierungsfunktion, oft gefolgt von Pooling. Typische Architektur für Bildklassifikation: mehrere Blöcke aus `Conv` $\rightarrow$ `ReLU` $\rightarrow$ `MaxPool`, wobei die räumliche Auflösung sinkt und die Anzahl der Kanäle steigt, dann *flatten* und ein oder zwei lineare Schichten mit `Softmax`. Die Faltungsschichten lernen in den frühen Layern allgemeine Merkmale wie Kanten und Ecken, in späteren Layern zusammengesetzte Muster.

Man beachte: Jede Berechnung mit einem Faltungslayer kann auch mit einem linearen (fully connected) Layer durchgeführt werden – die Faltung ist ja linear, also eine Matrix, nur eine sehr spezielle mit vielen Nullen und wiederholten Einträgen. Der Punkt der Faltung ist nicht, dass sie *mehr* kann, sondern dass sie mit viel weniger Parametern das Richtige kann, wenn die Daten lokale, verschiebungsinvariante Struktur haben. Und man muss nicht alle Layer zu Faltungen machen; die letzten Schichten eines CNN sind fast immer linear.

Ausgabegröße einer Faltungsschicht

Wie groß ist die Feature Map nach einer Faltung? Bei Eingabelänge n , Kernelgröße k , Padding p (Nullen an jedem Rand) und Stride s ist die Ausgabegröße

$$n_{\text{out}} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1, \quad (133)$$

pro Dimension. Ohne Padding („valid“) schrumpft das Bild also um $k - 1$; mit $p = (k - 1)/2$ („same“) bleibt es gleich groß. Pooling folgt derselben Formel mit $k = s = \text{Poolgröße}$. Beispiel für ein kleines CNN auf 28×28 -Bildern (MNIST):

```
nn.Conv2d(1, 8, kernel_size=5)      # 28 -> 24, 8 Kanäle
nn.MaxPool2d(kernel_size=2, stride=2) # 24 -> 12
nn.Conv2d(8, 16, kernel_size=5)     # 12 -> 8, 16 Kanäle
nn.MaxPool2d(kernel_size=2, stride=2) # 8 -> 4
nn.Flatten()                         # 16 * 4 * 4 = 256
nn.Linear(16 * 4 * 4, 10)
```

Die Zahl im `nn.Linear` muss man also ausrechnen – der häufigste Shape-Fehler beim Bau eines CNN. Mit `kernel_size=3` statt 5 wären es $28 \rightarrow 26 \rightarrow 13 \rightarrow 11 \rightarrow 5$, also $16 * 5 * 5$.

2-dimensionale Faltung in PyTorch

`nn.Conv2d` ist ein Layer mit lernbaren Gewichten; man kann die Gewichte aber auch von Hand setzen und den Layer als Bildfilter benutzen. Ein Farbfoto wird zu einem Tensor der Form `(1, 3, H, W)` (Batch, RGB-Kanäle, Höhe, Breite), und ein Filter mit `in_channels=3` summiert über die drei Farbkkanäle:

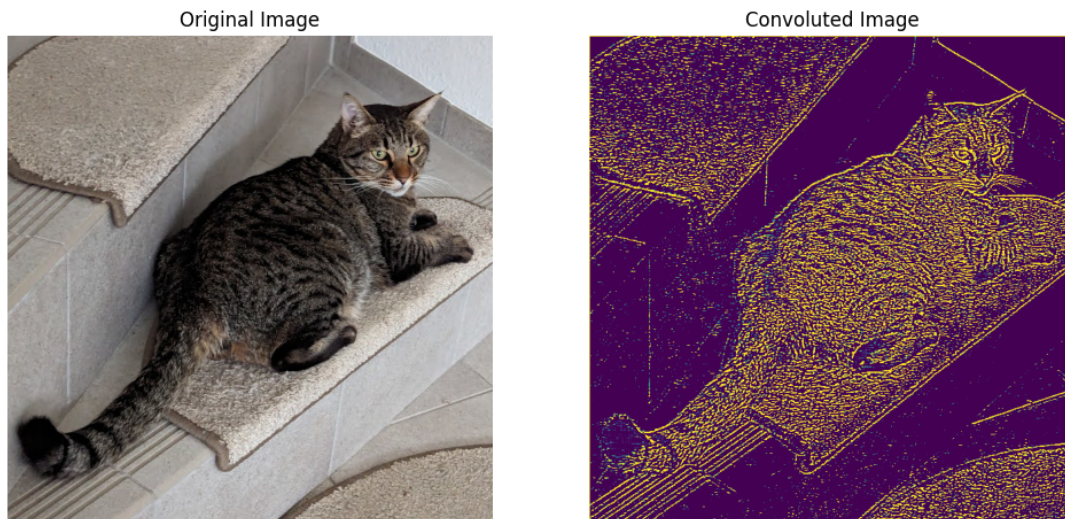
```
import torch, torch.nn as nn
from PIL import Image
from torchvision import transforms

img = transforms.ToTensor()(Image.open("pallino.jpg")).unsqueeze(0) #
↳ notebooks/pallino.jpg, (1, 3, H, W)

conv = nn.Conv2d(in_channels=3, out_channels=1, kernel_size=5, padding=1, bias=False)
w = torch.full((5, 5), -1.0)      # überall -1 ...
w[2, 1:4] = 7.0                  # ... außer drei Werten in der Mitte: 7
conv.weight.data = w.expand(1, 3, 5, 5).clone() # derselbe Filter auf allen Kanälen

out = torch.clamp(conv(img), 0, 1).squeeze()      # (H', W'), auf [0, 1] gekappt
```

Die Gewichte summieren sich zu $22 \cdot (-1) + 3 \cdot 7 = -1$: In gleichmäßigen Flächen ist die Antwort leicht negativ und wird auf 0 gekappt, nur dort, wo die drei Mittelpixel heller sind als ihre Umgebung – an Kanten und Texturen – bleibt etwas übrig. Das Ergebnis hat einen Kanal, ist also ein Graustufenbild; unten in Falschfarben.



`nn.Conv2d` mit dem handgesetzten 5×5 -Filter oben: Aus einem Foto wird ein Kantenbild. Ein CNN lernt solche Filter selbst – in der ersten Schicht sehen sie meist genau so aus.

Weiterführend:

- Animationen von Vincent Dumoulin und Francesco Visin zu Padding, Stride und Dilation: github.com/vdumoulin/conv_arithmetic.
- Interaktives MNIST-CNN-Explorationswerkzeug von Adam Harley: adamharley.com/nn_vis/cnn/2d.html – man zeichnet eine Ziffer und sieht alle Feature Maps aller Schichten.
- *Universal Fourier Features Emerge* ([arXiv:2312.08550](https://arxiv.org/abs/2312.08550)): warum gelernte Filter so oft wie Fourier-Basisfunktionen aussehen.
- Faltung und Pooling einmal von Hand zu rechnen lohnt sich; ein Übungsblatt tut genau das.

5.1.6 Übung

Signal $g = (3, 1, 4, 1, 5)$ an den Stellen $x = 0..4$, Filter $f = (\frac{1}{2}, 0, -\frac{1}{2})$ an den Stellen $-1, 0, 1$.

1. Berechnen Sie $(g * f)(x)$ für $x = 1, 2, 3$ mit $(g * f)(x) = \sum_k g(x - k)f(k)$. *Tipp: Nur drei Summanden je Stelle, und einer davon fällt weg.*
2. Berechnen Sie Max-Pooling von g mit Poolgröße 2 und Stride 2.
3. Ein 28×28 -Bild geht durch `Conv2d(1, 16, kernel_size=5)`, dann `MaxPool2d(2)`, dann `Conv2d(16, 16, kernel_size=5)`, dann `MaxPool2d(2)`. Welchen Shape hat die Ausgabe, und wie viele Werte hat sie nach **Flatten**? *Tipp: $\lfloor (n + 2p - k)/s \rfloor + 1$, hier ohne Padding.*

Antwortprobe: Bei (3) kommt $16 \cdot 4 \cdot 4$ heraus – die Zahl taucht in PyTorch-Beispielen ständig auf.

Weiterlesen

Prince, *UDL*, Kap. 10 (Convolutional networks) und 11 (Residual networks). Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).

5.2 LLMs benutzen

5.2.1 Transformer-LLMs in der Praxis einsetzen

Zur Erinnerung der Aufbau eines Transformers ([Kapitel „Transformer“](#), Fig. 1 in „Attention is all you need“): Encoder- und Decoder-Stapel aus Attention- und MLP-Blöcken. Ein *Large Language Model* ist ein sehr großer, meist Decoder-only Transformer, der das nächste Token vorhersagt.

Verschiedene Wege, LLMs zu nutzen

Wir konkretisieren anhand eines Beispiel-Tasks, z.B. „Gegeben einen Text, identifiziere, ob er emotional positiv oder negativ gefärbt ist“ (Sentiment-Analyse).

1. **Wort-Vektor-Einbettungen direkt verwenden** (statt word2vec z.B. die Embedding-Schicht von Gemma) → semantische Ähnlichkeit, semantische Suche, semantisches Clustering etc. Pro: sehr billig (keine tiefen Modelle für Inferenz notwendig). Contra: Zusammenhänge über längere Texte hinweg nicht sichtbar.
2. **Satz-Vektor-Einbettungen**, bzw. längere Texte in Vektoren umwandeln (z.B. mit BERT). Pro: sehr hohe Qualität der Repräsentationen. Contra: LLM notwendig zur Berechnung der Repräsentationen. (Das Notebook `latent-classifier.ipynb` – Download oben auf dieser Seite – baut genau das: BERT-Embeddings als Features für einen Klassifikator.)
3. **Nicht Token sampeln, sondern die Verteilung der Ausgab tokens direkt betrachten**. Pro: vollständige Information aus dem LLM, kein „zufälliges“ Sampling. Contra: weniger intuitiv zu verwenden, man arbeitet mit Verteilungen im Latent Space statt mit Text.
4. **Fertig trainierte Transformer mit bestimmten Prompts verwenden**, z.B. „Analog zur Transformation [A] → [B] verarbeite [C]“ (*one-shot prompt*), etwa [A] = Text auf Englisch, [B] = Text auf Deutsch, [C] = neuer Text auf Englisch. Pro: extrem vielseitig und bisher sehr, sehr erfolgreich. Contra: benötigt viele Token (noch mehr für *few-shot prompts*).
5. **Transformer von Grund auf trainieren** (eigenes Pretraining). Pro: vollständige Kontrolle über Trainingsdaten und damit prinzipiell über Outputs. Contra: extrem teuer; Llama 2 (etwa auf dem Niveau von Gemini 1 Pro / ChatGPT 3.5) kostete ca. 2–10 Mio. \$.
6. **Vortrainierte Transformer auf besonderen Daten weiter trainieren** (Finetuning). Pro: relativ billig (keine sehr langen Trainingszeiten = Stromkosten nötig). Contra: benötigt wenigstens Hardware zum Training (teurer für größere Modelle).
7. **Transformer-Modell auf Chat-Formate finetunen mit RLHF** (Reinforcement Learning from Human Feedback, wie bei ChatGPT). Pro: Feedback führt zu Qualitätssteigerung, z.B. NSFW-Removal o.ä. Contra: etwas komplizierter als „einfaches“ Training.
8. **Sehr große Modelle über eine API verwenden** (ggf. nur Sampling-Output). Der De-facto-Standard für die „meisten“ LLM-Use-Cases für Softwareentwickler.
9. **Vorhersagen eines großen Modells in ein kleineres übertragen** (Distillation). Das schauen wir uns im [Kapitel „Distillation“](#) noch an; so lässt sich z.B. die Inferenz beschleunigen.
10. **Constrained Generation**: Ausgabe sampeln, die einem (z.B. JSON-)Schema genügt – ein wesentlicher Baustein für „agentische Workflows“.

5.2.2 Selbst Inferenz und Training durchführen

Rechnung mit der Parametermenge

Hugging Face Transformers ist die Standardbibliothek; die Faustrechnung geht so. Allein die **Gewichte** von Llama 2 7B belegen bei 32 Bit pro Parameter $7 \cdot 4 \text{ GB} = 28 \text{ GB}$, bei 16 Bit $7 \cdot 2 \text{ GB} = 14 \text{ GB}$ und mit 4-Bit-**Quantisierung** noch $7 \cdot \frac{1}{2} \text{ GB} \approx 3,5 \text{ GB}$. Für **Inferenz** kommt darüber hinaus nur wenig dazu (Aktivierungen und der KV-Cache), 16 Bit passen also auf eine 16-GB-Consumer-GPU und 4 Bit auf fast alles. Beim **Training** dagegen braucht jeder Parameter zusätzlich einen Gradienten und die Zustände des Optimierers – bei Adam zwei Momente, zusammen grob das Vierfache der Gewichte. Nicht die Inferenz ist also das Problem auf Consumer-Hardware, sondern das Training. Llama 2 70B ist erst in einer Liga mit GPT-4, und dessen Gewichte allein sind 35 GB (4 Bit) bis 280 GB (32 Bit).

Llamas & Co

- Llama (1, 2, 3) sind *open-weight*-Modelle mit verschiedenen Lizenzen – die Gewichte sind frei verfügbar, die Trainingsdaten nicht.
- Llama 2 7B ist als fp16-Download 13,5 GB groß (das sind die $7 \cdot 2$ GB von oben); als 4-Bit-GGUF sind es gut 4 GB, und damit passt es auf eine einfache Consumer-Grafikkarte.
- Llama3.3-70B braucht sehr viel VRAM oder rechnet langsam, ist dafür „in derselben Liga“ wie GPT-4.
- Llama3.1-405B verteilt man eher auf mehrere GPUs (vgl. auch GPT-3 mit ca. 175B Parametern).

Llamas benutzen

Sofort ausprobieren mit **Ollama** ([Quickstart](#)): startet einen Webservice mit Chat-Interface und lädt Modelle selbst herunter. Ollama baut auf **llama.cpp** auf, bzw. auf Georgi Gerganovs **ggml**-Bibliothek, einer Transformer-Architektur-optimierten Deep-Learning-Bibliothek in C/C++ – die auf CPUs erstaunlich schnell ist. Eigenes Dateiformat **GGUF** für (quantisierte) Modelle. Achtung: Nicht immer sind die Chat/RLHF/Instruct-Modelle die geeigneten LLMs (z.B. für Code-Autocompletion à la Copilot oder RAG nimmt man eher die Basismodelle).

```
curl -fsSL https://ollama.com/install.sh | sh
ollama run llama3.2:1b "Explain how LLMs work"
```

Das ist kein Deep Learning, das ist „Deep-Learning-Produkte benutzen“ – trotzdem super für Rapid Prototyping und jede Menge Anwendungen. Ollama bietet einen OpenAI-kompatiblen API-Endpoint, das verhindert Vendor Lock-in: Code, der gegen die OpenAI-API geschrieben ist, läuft mit einer Zeile Änderung lokal.

Lokale LLMs benutzen: Wege 2, 3 und 10 in Code

Wir probieren die Wege 2, 3 und 10 mit `qwen2.5:0.5b` aus – knapp 400 MB, läuft auf der Laptop-CPU. Vorbereitung im Terminal:

```
ollama pull qwen2.5:0.5b
ollama run qwen2.5:0.5b      # interaktiver Chat, mit /bye beenden
pip install ollama llama-cpp-python pydantic
```

Ein erster Test im Chat: „Wie kann man mit Deep Learning Hunde von Katzen unterscheiden (also, warum funktioniert das)? Zähle nur 5–10 Stichpunkte auf.“ Lesen und amüsieren – 0,5 Milliarden Parameter sind eben nicht 50 Milliarden; auf Englisch fällt die Antwort meist besser aus. Ollama läuft danach als Server auf Port 11434 (`http://localhost:11434` zeigt „Ollama is running“, unter `/v1/models` steht das Modell).

Weg 2, Embeddings. Ein Text wird zu einem Vektor fester Länge (bei Qwen 2.5 0.5B: 896 Dimensionen), ideal für semantische Suche oder als Feature für einen Klassifikator:

```
import ollama, numpy as np

response = ollama.embeddings(model="qwen2.5:0.5b", prompt="Heute ist es kalt
↳ draußen.")
vector = np.array(response["embedding"])
print(vector.shape, np.linalg.norm(vector))      # (896,) ~300
```

Weg 3, die Verteilung des nächsten Tokens. Statt Text zu generieren, schauen wir uns die Logits an. Dafür nehmen wir `llama.cpp` direkt (den Python-Wrapper `llama_cpp`) und das Modell als GGUF-Datei von huggingface.co/Qwen/Qwen2.5-0.5B-Instruct-GGUF (Variante `q4_k_m`, 491 MB):

```
from llama_cpp import Llama
import numpy as np

llm = Llama(model_path="qwen2.5-0.5b-instruct-q4_k_m.gguf", logits_all=True,
↳ n_ctx=512, verbose=False)
```

```
def next_token_probs(text):
    llm.reset()
    tokens = llm.tokenize(text.encode("utf-8"))
    llm.eval(tokens)
    logits = np.array(llm.scores[len(tokens) - 1]) # Logits an der letzten Position
    return np.exp(logits) / np.sum(np.exp(logits)) # Softmax

probs = next_token_probs("The color of the sky is")
print(probs.shape) # (151936,) -- die Vokabulargröße
for i in np.argsort(probs)[-5:][::-1]:
    print(repr(llm.detokenize([i]).decode()), f"{probs[i]:.2%}")
# ' changing' 10.3%, ' always' 6.7%, ' blue' 4.0%, ' a' 3.2%, ' determined' 3.0%
red = llm.tokenize(b" red")[0]
print(f"{probs[red]:.2%}") # 0.72%
```

Das ist die logistische Regression aus dem [Kapitel „Logistische Regression“](#) – Softmax über Logits –, nur mit 150 000 Klassen; alles aus dem [Kapitel „Latente Räume geometrisch“](#) über Sampling-Strategien und Temperatur setzt genau hier an. (Bei `llm.scores` gibt es einen Fallstrick: Das Array hat immer `n_ctx` Zeilen, `scores[-1]` wäre also die falsche Position.)

Weg 10, strukturierter Output. Wir zwingen das LLM, JSON auszugeben, und nutzen `pydantic`, um zu garantieren, dass die Datentypen stimmen:

```
from pydantic import BaseModel
import ollama

class Person(BaseModel):
    name: str
    age: int
    occupation: str
    location: str

text = "Yesterday I met Sabine. She is 29 years old, works as a doctor and loves her
↪ life in Munich."
response = ollama.chat(
    model="qwen2.5:0.5b", format="json",
    messages=[{"role": "user", "content":
        f"Extract information about the person from this text into JSON using
        ↪ {list(Person.model_fields)}: {text}"},
    ])
person = Person.model_validate_json(response["message"]["content"])
print(person) # name='Sabine' age=29 occupation='doctor' location='Munich'
```

`format="json"` ist *constrained generation*: Beim Sampling werden nur Token zugelassen, die die Ausgabe syntaktisch gültig halten. Die Validierung mit `pydantic` prüft danach, ob auch die Typen stimmen (`age` ist wirklich ein `int`) – erst damit wird die Ausgabe eines LLM zu etwas, worauf man Software bauen kann.

Selbst Transformer finetunen

Finetuning ist auch sehr teuer, weil alle Parameter Gradienten und Optimizer-Zustände brauchen. Lösung: **Parameter-Efficient Finetuning** (PEFT) mit **Low-Rank Adaptations** (LoRA, [Conceptual Guide](#)): Die Gewichtsmatrizen W bleiben eingefroren, und man lernt nur eine kleine Korrektur $\Delta W = AB$ mit $A \in \mathbb{R}^{n \times r}$, $B \in \mathbb{R}^{r \times m}$ und Rang $r \ll n, m$. Statt $n \cdot m$ Parametern sind das $r(n + m)$ – für $r = 8$ oft unter einem Prozent des Modells.

5.2.3 Wie geht eigentlich ...eine Anwendung?

Konkret eine Anwendung wie z.B. der „Chatbot“ [hAInrich](#) am HeiCAD: Retrieval-Augmented Generation (RAG) – Dokumente werden mit Weg 2 in Vektoren verwandelt und in einer Vektordatenbank abgelegt, zur Anfrage werden die ähnlichsten Dokumente gesucht und mit Weg 4 als Kontext in den Prompt eines LLM gegeben. Mehr dazu im [Kapitel „Anwendungsbeispiele“](#).

5.2.4 Ausblick: Transfer Learning und Ensembles

Wir schauen rein, wenn dazu noch Zeit bleibt: das Kaggle-Notebook „PyTorch | Flowers | TransLearning + Ensemble | Test = 99.67 %“. Fragen, die wir klären wollen: Worum geht es da überhaupt? Was „ist“ Ensembling bzw. ein Ensemble-Modell? Was ist z.B. das Modell GoogLeNet von 2014, das da verwendet wird?

5.3 Generative Adversarial Networks

5.3.1 Historie

NeurIPS 2014

Goodfellow et al., *Generative Adversarial Nets*: „We propose a new framework for estimating generative models via adversarial nets, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G . The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D , a unique solution exists, with G recovering the training data distribution and D equal to $1/2$ everywhere. In the case where G and D are defined by multilayer perceptrons, the entire system can be trained with backpropagation.“

Der letzte Satz ist der Punkt: Kein Markov-Chain-Sampling, keine Variationsrechnung wie beim VAE – nur zwei Netze und Backpropagation.

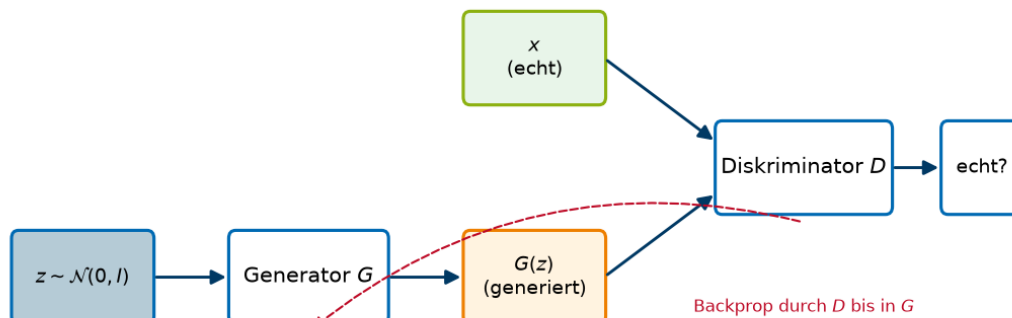
LeCun 2016

Yann LeCun, auf Quora: „There are many interesting recent developments in deep learning [...] The most important one, in my opinion, is adversarial training (also called GAN for Generative Adversarial Networks). This is an idea that was originally proposed by Ian Goodfellow when he was a student with Yoshua Bengio at the University of Montreal [...] This, and the variations that are now being proposed is the most interesting idea in the last 10 years in ML, in my opinion.“

5.3.2 GANs technisch

Aufbau

Zwei Netze: Der **Generator** G bekommt einen zufälligen latenten Vektor $z \sim \mathcal{N}(0, I)$ (das *Rauschen*) und erzeugt daraus ein Sample $G(z)$, z.B. ein Bild. Der **Diskriminator** D bekommt ein Bild – entweder ein echtes x aus dem Trainingsdatensatz oder ein generiertes $G(z)$ – und gibt eine Wahrscheinlichkeit $D(x) \in (0, 1)$ aus, dass das Bild echt ist. Man beachte: Der Generator bekommt *nie* echte Bilder als Input; er sieht die Daten nur indirekt, über den Gradienten des Diskriminators.



Generator und Diskriminator. Der Generator sieht nie echte Daten – nur den Gradienten, der durch den Diskriminator zurückfließt.

Erinnerung: Samples generieren

Der Generator generiert Samples aus einer Verteilung. Diese Verteilung ist, was im (Pre-)Training gelernt wird, indem die Samples aus dem Trainingsdatensatz als latente Vektoren repräsentiert werden. Generatives Modell = Verteilung lernen ([Kapitel „Was ist Deep Learning?“](#)). Beim VAE haben wir dafür den latenten Raum explizit auf eine Normalverteilung gezwungen; beim GAN sagt uns der Diskriminator, ob die generierte Verteilung schon von der echten zu unterscheiden ist.

GAN-Training

Das *Minimax-Spiel* aus dem Abstract als Formel:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim \mathcal{N}(0, I)} [\log(1 - D(G(z)))]. \quad (134)$$

Das ist die **negative** binäre Kreuzentropie des Diskriminators mit Label 1 für echt und 0 für generiert – in der Schreibweise aus dem [Kapitel „Loss-Funktionen und Maximum Likelihood“](#) also $-L_{\text{BCE}}$, ohne den Faktor $\frac{1}{N}$. Deshalb *maximiert* D sie – gut klassifizieren heißt ja, die Kreuzentropie klein zu machen –, und G minimiert sie (D täuschen). In der Praxis werden beide abwechselnd mit Gradientenabstieg trainiert:

Diskriminator: Er bekommt eine Penalty in der Loss-Funktion für die Fehlklassifikation eines Eingabebildes, also sowohl für falsch positive („echt“ behauptet, war aber generiert) als auch falsch negative („generiert“ behauptet, war aber echt). Ein Schritt: ein Batch echter Bilder mit Label 1, ein Batch generierter Bilder mit Label 0, BCE-Loss, **backward, step** – nur die Parameter von D .

Generator: Nach einem optionalen Pretraining (z.B. als Decoder eines Autoencoders, um die Verteilung der Trainings-Samples grob zu lernen) wird der Generator zusammen mit dem Diskriminator trainiert (Finetuning). Dazu bekommt der Generator eine Penalty in der Loss-Funktion für das Generieren von Bildern, die der Diskriminator als generiert erkennt: ein Batch $G(z)$, durch D schicken, BCE-Loss mit Label 1 (der Generator will, dass D „echt“ sagt), **backward, step** – nur die Parameter von G . Der Gradient fließt dabei durch D hindurch bis in G , das ist Backpropagation durch zwei Netze.

Im Gleichgewicht, wenn G die Datenverteilung exakt trifft, kann D nicht besser als raten: $D \equiv \frac{1}{2}$.

GAN-Modelle

Der **Diskriminator** kann ein einfaches Multilayer Perceptron sein, d.h. mehrere lineare Layer und zuletzt ein einzelnes Neuron zur Klassifikation „ist das echt?“ in 0 oder 1 – also logistische Regression auf Features. Alternativ kann der Diskriminator auch aus 2 Komponenten bestehen, wobei zunächst ein vor-trainierter Encoder das Bild in latente Vektoren umrechnet, die dann an einen Klassifikator weitergegeben werden. Dabei kann z.B. ein CNN zur Extraktion von Features sinnvoll sein.

Der **Generator** kann ein einfaches Multilayer Perceptron sein, etwa der Decoder eines Autoencoders, sodass es einen latenten Raum gibt, aus dem man sampeln kann. Mit Faltungen (genauer: *transponierten* Faltungen, die die Auflösung erhöhen) im Generator und einem CNN im Diskriminator heißt das **DCGAN** (deep convolutional GAN, Radford et al. 2015).

Ausprobieren: GAN Lab

Ausprobieren: GAN Lab

poloclub.github.io/ganlab (2019, Georgia Tech und Google Brain): ein GAN auf 2D-Punktwolken im Browser, mit der Generator-Verteilung, dem Gradientenfeld des Diskriminators und den Loss-Kurven live. Was man dort sehen sollte: Bei der Ring- oder Gauß-Wolke wandert die generierte Verteilung zur echten, bis das Gradientenfeld flach wird. Beim *linearen* Datensatz läuft das Training regelmäßig in ein lokales Minimum, in dem der Generator nur einen Teil der Linie bedient, und mehrere Neustarts liefern verschiedene Ergebnisse; das ist Mode Collapse in klein. Daraus folgt die Faustregel für GANs: *früh aufhören*. Anders als beim Autoencoder wird es nicht monoton besser. Man beobachtet die Samples und behält die Gewichte von dem Zeitpunkt, an dem sie gut aussahen. Die Loss-Kurven sind dabei nur bedingt hilfreich, siehe unten bei Wasserstein.

5.3.3 Varianten von GANs

Conditional GANs

Generator und Diskriminator bekommen Zusatzinformationen, z.B. Klassenlabels: Der Generator generiert $G(z, y)$ bedingt auf ein gegebenes Klassenlabel y , der Diskriminator entscheidet $D(x, y)$ bedingt zum Klassenlabel. So bekommt man einen Generator, dem man sagen kann, *welche* Ziffer er zeichnen soll (die vierte MNIST-Task aus dem [Kapitel „Autoencoder“](#)). Geht auch mit anderen Zusatzinformationen, z.B. Text kodiert als Vektor von Wort-Embeddings – das ist der Weg zu Text-zu-Bild-Modellen.

Wasserstein GANs – wozu?

Probleme mit klassischen GANs bei der Bildgenerierung:

- **Mode Collapse** – der Generator produziert nur (noch) stereotype Bilder ohne große Variation. Die Verteilung kollabiert auf die Moden, also die Wahrscheinlichkeitsmasse konzentriert sich auf einige wenige Punkte, die D gerade nicht erkennt.
- **Vanishing Gradients** – wenn der Diskriminator zu schnell gut wird, ist $D(G(z)) \approx 0$ überall, $\log(1 - D(G(z)))$ ist flach, und der Lernprozess geht für den Generator nicht weiter.
- **Instabilität im Training** – die Loss-Funktion des Generators entspricht nicht unbedingt der Qualität der Bilder; man kann am Loss nicht ablesen, ob es besser wird.

Wasserstein GANs – was?

Andere Loss-Funktion: die **Wasserstein-Distanz**, auch „Earth Mover’s Distance“ – wie viel Wahrscheinlichkeitsmasse (mal Weglänge) bewegt werden muss, um aus einer Verteilung eine andere zu machen (analog zur Levenshtein-Distanz als Editierabstand für Strings). Die Verteilungen sind zum einen die echten Bilder, zum anderen die generierten Bilder (je ein Batch). Der Vorteil gegenüber der KL-Divergenz bzw. der Kreuzentropie: Die Wasserstein-Distanz ist auch dann endlich und hat einen nützlichen Gradienten, wenn sich die beiden Verteilungen gar nicht überlappen.

Kritiker statt Diskriminator: Statt einer Wahrscheinlichkeit in $(0, 1)$ gibt der Kritiker eine reelle Zahl aus und schätzt damit die Wasserstein-Distanz zwischen den realen Bildern und den gegebenen Bildern. Wenn der Kritiker alle Bilder für echt hält, schätzt er 0, ansonsten gibt es ein Signal, welches für das Training genutzt werden kann. Beides führt zu glatteren Loss-Landschaften und damit zu effektiverem Gradientenabstieg, und der Kritiker-Loss korreliert tatsächlich mit der Bildqualität.

Vertiefung: warum der Kritiker die Wasserstein-Distanz schätzt

Die Wasserstein-1-Distanz zwischen zwei Verteilungen p und q ist als Transportproblem definiert. Die Kantorovich-Rubinstein-Dualität sagt, dass man sie auch so schreiben kann:

$$W(p, q) = \sup_{\|f\|_{\text{Lip}} \leq 1} \left(\mathbb{E}_{x \sim p}[f(x)] - \mathbb{E}_{x \sim q}[f(x)] \right), \quad (135)$$

ein Supremum über alle Funktionen f , die höchstens Steigung 1 haben (1-Lipschitz). Der Kritiker ist dieses f : ein Netz, das auf echten Bildern hohe und auf generierten Bildern niedrige Werte ausgeben soll; die Differenz der Mittelwerte ist die geschätzte Distanz. Damit das eine Schranke bleibt, muss der Kritiker Lipschitz-beschränkt bleiben: im Original-WGAN (Arjovsky et al. 2017) durch Abschneiden der Gewichte (*weight clipping*), in WGAN-GP (Gulrajani et al. 2017) durch einen Strafterm auf die Gradientennorm (*gradient penalty*). Tutorial: Lilian Weng, [From GAN to WGAN](#).

5.3.4 Konkrete Modelle

CycleGAN

Zhu, Park, Isola, Efros (ICCV 2017), *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*: Image-to-image translation lernt die Abbildung von einem Eingabebild auf ein Ausgabebild – normalerweise aus einem Trainingsdatensatz *gepaarter* Bilder. Für viele Tasks gibt es aber keine Paare (Foto $\leftrightarrow$ Monet-Gemälde, Pferd $\leftrightarrow$ Zebra, Sommer $\leftrightarrow$ Winter). CycleGAN lernt eine Abbildung $G: X \rightarrow Y$, sodass die Verteilung von $G(X)$ von Y nicht unterscheidbar ist (adversarial loss) – und weil das allein hochgradig unterbestimmt ist, gleichzeitig die inverse Abbildung $F: Y \rightarrow X$ mit einem **Cycle-Consistency-Loss** $F(G(x)) \approx x$ (und umgekehrt). Ein Pferd, das zum Zebra und zurück wird, muss wieder dasselbe Pferd sein.

Beispielbilder (Monet $\leftrightarrow$ Foto, Zebra $\leftrightarrow$ Pferd, Sommer $\leftrightarrow$ Winter) auf der Projektseite [junyanz.github.io/CycleGAN](#).

StyleGAN

Karras, Laine, Aila – NVIDIA (CVPR 2019), *A Style-Based Generator Architecture for Generative Adversarial Networks*: eine alternative Generator-Architektur, die sich an der Style-Transfer-Literatur orientiert. Der latente Vektor z wird zuerst durch ein MLP in einen Zwischenraum w abgebildet und

dann *auf jeder Auflösungsstufe* des Generators als „Stil“ eingespeist. Das führt zu einer automatisch gelernten, unsupervised Trennung von High-Level-Attributen (z.B. Pose und Identität bei Gesichtern) und stochastischer Variation (z.B. Sommersprossen, Haare), und erlaubt intuitive, skalenspezifische Kontrolle der Synthese. Dazu ein neuer Datensatz von Gesichtern (FFHQ). StyleGAN2 ist hinter [thispersondoes-notexist.com](https://thispersondoesnotexist.com).

Beispielgesichter und das Video zur Interpolation im Stil-Raum: github.com/NVlabs/stylegan.

Viele konkrete Modelle

The GAN Zoo, github.com/hindupuravinash/the-gan-zoo: eine Liste von mehreren hundert benannten GAN-Varianten. Seit ca. 2021 haben Diffusionsmodelle die GANs bei der Bildgenerierung weitgehend abgelöst, aber das adversarielle Prinzip lebt weiter – z.B. als Loss-Term beim Training von VAE-Decodern für Stable Diffusion.

5.3.5 Selbst GANs basteln

Tutorial time: pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html, das offizielle DCGAN-Tutorial auf dem CelebA-Gesichterdatensatz. Die Trainingsschleife dort ist genau die oben beschriebene: erst D mit echten und generierten Bildern, dann G mit Label „echt“, abwechselnd pro Batch.

Weiterlesen

Prince, *UDL*, Kap. 15 (Generative adversarial networks), inklusive Wasserstein-GAN und StyleGAN. Das Buch ist frei verfügbar, siehe [Über dieses Skript](#).

5.4 Latente Räume geometrisch

Autoencoder, VAE, word2vec, GANs – alle diese Modelle haben einen *latenten Raum*, in dem die gelernten Repräsentationen leben. In diesem Kapitel geht es um die Geometrie dieser Räume: Was bedeuten Richtungen und Abstände, was passiert zwischen zwei Codes, und was unterscheidet den diskreten latenten Raum eines Sprachmodells vom kontinuierlichen eines Bild-Autoencoders?

5.4.1 Wortvektoren geometrisch verstehen

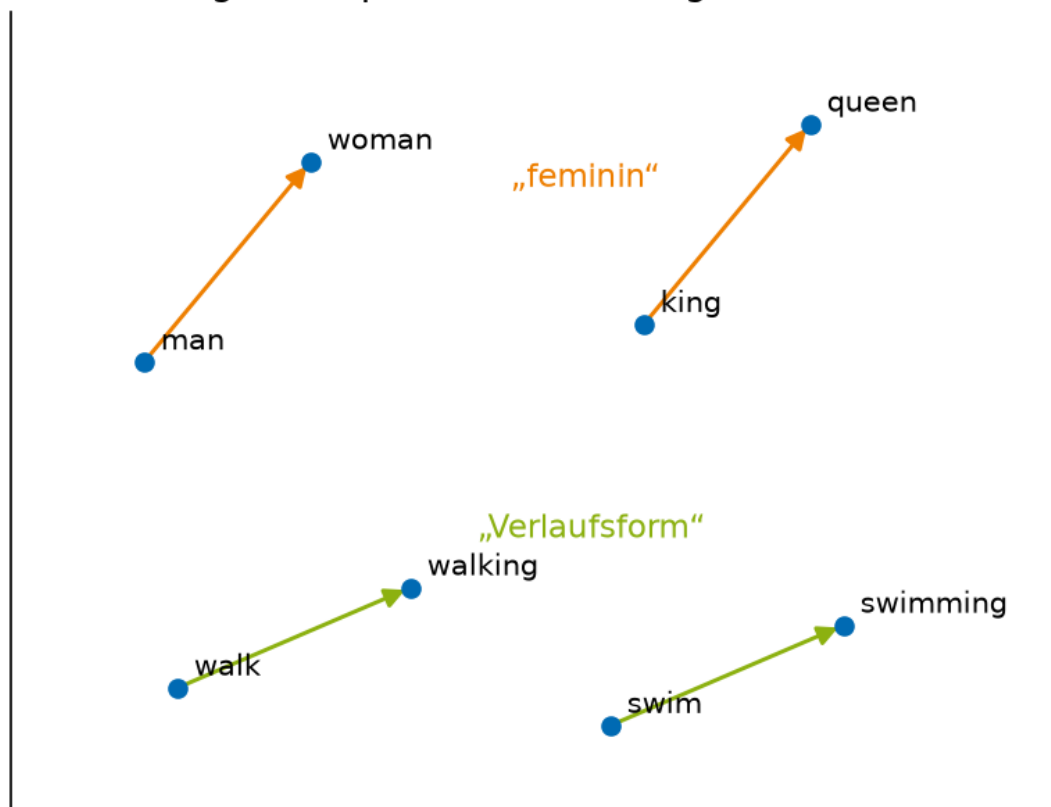
Arithmetik im Latent Space

Der Klassiker bei word2vec: $\text{KING} - \text{MAN} + \text{WOMAN} = \text{QUEEN}$. Also ist $\text{WOMAN} - \text{MAN}$ die Richtung „feminin“. Auch:

- $\text{PARIS} - \text{FRANCE} + \text{ITALY} = \text{ROME}$ (also $\text{PARIS} - \text{FRANCE}$ die Richtung „Hauptstadt“),
- $\text{WALKED} - \text{WALK} + \text{SWIM} = \text{SWAM}$ (also $\text{WALKED} - \text{WALK}$ die Richtung „Vergangenheit“),
- $\text{BIGGER} - \text{BIG} = \text{BIGGEST} - \text{BIGGER}$ (also Steigerung der Intensität als Richtung).

Analog, aber in allen LLMs: Semantische Beziehungen werden zu *Richtungen* im latenten Raum, und Richtungen kann man addieren. Das ist keine Eigenschaft, auf die word2vec trainiert wurde, sie entsteht von selbst aus dem Kontext-Vorhersage-Training.

Beziehungen als parallele Richtungen (schematisch)



Beziehungen zwischen Wörtern als parallele Richtungen im Vektorraum (schematisch).

Bolukbasi et al. 2016, *Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings*: Dieselbe Arithmetik findet auch $\text{MAN} - \text{WOMAN} \approx \text{COMPUTER PROGRAMMER} - \text{HOMEMAKER}$. Die Embeddings enthalten die Stereotype des Trainingskorpus als Richtungen, und man kann sie messen – und, das ist der Beitrag des Papers, durch Projektion auf das orthogonale Komplement der Gender-Richtung teilweise entfernen (Abbildung oben).



Figure 7: Selected words projected along two axes: x is a projection onto the difference between the embeddings of the words *he* and *she*, and y is a direction learned in the embedding that captures gender neutrality, with gender neutral words above the line and gender specific words below the line. Our hard debiasing algorithm removes the gender pair associations for gender neutral words. In this figure, the words above the horizontal line would all be collapsed to the vertical line.

Figure 7 aus Bolukbasi et al. 2016 – Berufe, projiziert auf die Gender-Richtung.

Interpolation und Superposition

Ein Vektor im Latent Space ist ausdrucksstärker als ein Token, er kann interpolieren und Superposition ausdrücken: Statt „Katze“ oder „Hund“ geht auch $\frac{1}{2}$ Katze + $\frac{1}{2}$ Hund oder auch $t \cdot \text{Katze} + (1 - t) \cdot \text{Hund}$. Aber: $\text{decode}(t \cdot \text{Katze} + (1 - t) \cdot \text{Hund})$ ist entweder Katze oder Hund, es gibt kein „dazwischen“-Token außerhalb des Vokabulars.

Diskreter Latent Space

Bei word2vec und LLMs mit einem Vokabular von N Token können nur genau N Vektoren im Latent Space direkt zu Token dekodiert werden; für jeden anderen Vektor werden nahe liegende solche Vektoren ausgewählt zum Dekodieren. Gegeben ein Vektor im Latent Space, welches Token wählen wir aus? Das ist die **Sampling-Strategie**.

Gegeben Vektor $v \in \mathbb{R}^n$, welches Token $t \in \text{Vokabular}$ wählen wir?

- Berechne für jedes t den Abstand $d(v, \text{code}(t))$, z.B. im Kosinus-Abstand $1 - \cos(\angle(v, \text{code}(t)))$, und wähle t , sodass dieser Abstand minimal wird (**top-1**, *greedy decoding*).
- Betrachte auf allen Token t die Verteilung $\text{Softmax}(-d(v, \text{code}(t)))$ und **sample** aus dieser Verteilung.
- Für eine natürliche Zahl k (z.B. $k = 5$) wähle die k nächstgelegenen Token $t_1, \dots, t_k$ aus und sample mit der Softmax-Verteilung nur aus $\{t_1, \dots, t_k\}$ (**top-k sampling**; *top-p* nimmt stattdessen so viele Token, bis ihre Wahrscheinlichkeit p überschreitet).
- Verändere die Verteilung: Richtung top-1 durch niedrige Temperatur, Richtung Gleichverteilung durch hohe Temperatur.

Temperature Scaling: Gegeben eine Temperatur $T > 0$, berechne die Verteilung

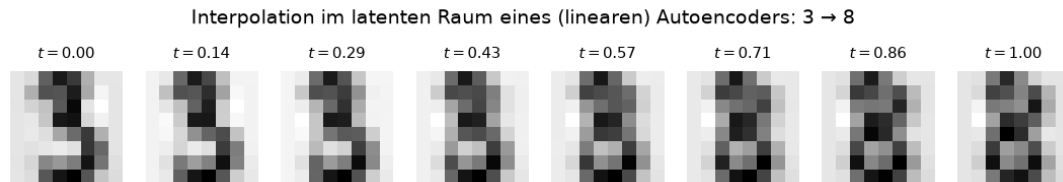
$$P_i = \frac{\exp(v_i/T)}{\sum_j \exp(v_j/T)}, \quad (136)$$

d.h. die Einträge von v (die Logits) werden durch T geteilt, dann wird mit Softmax eine Wahrscheinlichkeitsverteilung daraus. Großes $T > 1$: Die verschiedenen P_i werden sich ähnlicher, das Modell wird „kreativer“ (und fehleranfälliger). Kleines $T < 1$: Die großen P_i werden größer, die kleinen P_i werden kleiner; im Grenzfall $T \rightarrow 0$ ist es top-1.

5.4.2 Latente Räume für Bilder: AE, VAE, GAN

Erinnerung: kontinuierlich statt diskret

Bei word2vec und LLMs können nur N Vektoren dekodiert werden. Bei Autoencodern für Bilder ist dies anders: *Jeder* Vektor im Latent Space lässt sich zu einem Bild dekodieren. Damit lässt sich z.B. zwischen Hund und Katze interpolieren – der Decoder produziert für $t \cdot \text{enc}(\text{Hund}) + (1 - t) \text{enc}(\text{Katze})$ tatsächlich ein Bild, ob es sinnvoll aussieht, ist eine andere Frage.



Lineare Interpolation zwischen den Codes einer 3 und einer 8 im latenten Raum eines linearen Autoencoders (PCA mit 16 Komponenten auf den 8×8-Ziffern aus scikit-learn): Jeder Zwischenpunkt ist ein Bild.

Autoencoder vs. GANs

Der Generator eines GANs ist typischerweise gegeben durch eine Verteilung auf einem Latent Space, aus der gesampelt wird (i.d.R. eine Standardnormalverteilung), und einen Decoder, der aus dem latenten Vektor ein tatsächliches Bild generiert. Um einen guten Decoder zu bekommen, ist vor dem adversarial Training ein Pretraining in einem Autoencoder-Setup sinnvoll; daher ist der Latent Space zunächst derselbe. Nach dem GAN-Training des Decoders passen Encoder und Decoder aber nicht mehr zusammen, d.h. der Rekonstruktions-Loss steigt, während der Adversarial Loss sinkt. Der GAN-Decoder optimiert „sieht echt aus“, nicht „ist das Original“.

Autoencoder vs. VAE

Der Encoder eines VAE kodiert ein Eingabebild x als Paar (μ_x, σ_x) , wobei $\mu_x \in \mathbb{R}^n$ ein latenter Vektor wie beim Autoencoder ist. Der Unterschied vom VAE zum AE liegt im Training: Statt $\text{decode}(\mu_x)$ mit x zu vergleichen, wird $\text{decode}(\mu_x + \varepsilon)$ mit x verglichen, wobei ε Varianz σ_x^2 hat. Nach erfolgreichem Training ist dann auch $\text{decode}(\mu_x) \approx x$, aber der Latent Space bzw. Decoder sieht anders aus als beim AE: Um jeden Code herum gibt es eine Umgebung, die ebenfalls zu einem ähnlichen Bild dekodiert.

Häufig wird zusätzlich zum $+\varepsilon$ -Trick auch noch ein Summand zur Loss-Funktion hinzugefügt, damit die Codes der Trainingsdaten im Latent Space ungefähr standardnormalverteilt sind (Variational Loss, siehe [Kapitel „Variational Autoencoder“](#)). So kann man später aus der Standardnormalverteilung im Latent Space sampeln und bleibt in der gelernten Verteilung.

Arvanitidis et al. 2017: Latent Space Oddity

Latent Space Oddity: on the Curvature of Deep Generative Models (ICLR 2018): Der latente Raum eines VAE sieht flach aus, aber der Decoder verzerrt ihn – die euklidische Gerade zwischen zwei Codes ist nicht der kürzeste Weg im *Bildraum*. Die richtige Geometrie ist die durch den Decoder induzierte Riemannsche Metrik, und Geodäten darin laufen um Regionen herum, in denen der Decoder unsicher ist (Abbildung unten). Praktisch: Interpolationen entlang Geodäten sehen besser aus als lineare.

5.4.3 Übung

Gegeben Embeddings KÖNIG = (4, 2), MANN = (3, 0), FRAU = (1, 1).

1. Berechnen Sie KÖNIG – MANN + FRAU. Welcher der Vektoren $a = (4, 6)$, $b = (2, 2)$, $c = (0, 4)$ liegt am nächsten – gemessen mit dem Kosinus? *Tipp: Kosinus heißt: normieren, dann Skalarprodukt. Die Längen unterscheiden sich hier deutlich, das Ergebnis mit euklidischem Abstand wäre ein anderes.*
2. Interpolieren Sie linear zwischen MANN und FRAU in drei Schritten.

Antwortprobe: Bei (1) sind die Antworten für Kosinus und euklidischen Abstand verschieden – das ist der Punkt der Aufgabe.

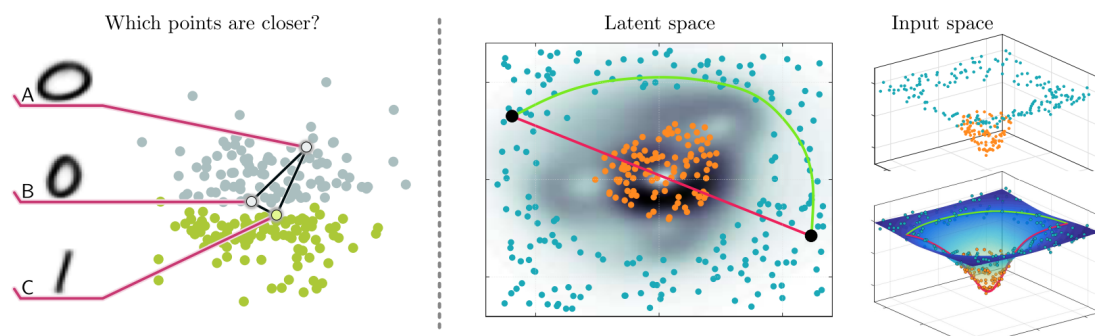


Figure 1: *Left*: An example of how latent space distances do not reflect actual data distances. *Right*: Shortest paths on the surface spanned by the generator do not correspond to straight lines in the latent space, as is assumed by the Euclidean metric.

Figure 1 aus Arvanitidis, Hansen, Hauberg: *Latent Space Oddity* (ICLR 2018): Die Gerade im latenten Raum (links) und die Geodäte bezüglich der vom Decoder induzierten Metrik führen zu unterschiedlichen Interpolationen.

5.5 Anwendungsbeispiele

Drei Beispiele, wie sich die Bausteine aus den vorigen Kapiteln zu Anwendungen zusammensetzen – und ein Blick darauf, wie man mit generiertem Code arbeitet.

5.5.1 Drei Beispiele aus der Praxis

Beispiel: hAInrich-Chatbot

hAInrich: Get answers quickly. Die Anwendung ist ein Interface zu einer Wissensdatenbank, eine semantische Suche im Gewand eines Chatbots. Das Wissen liegt vor als Frage-Antwort-Pärchen-Liste (handgeschrieben: FAQ). Ziel: eine beliebige Frage „korrekt“ beantworten. Nebenbedingung: kein Logging (DSGVO), geringe Last auf dem Server (Kosten). Erreichbares Ziel: oft korrekt raten, welche der handgeschriebenen Fragen (mit vorhandenen Antworten) ungefähr der eingegebenen Nutzerfrage entspricht.

Wir überlegen gemeinsam: Wie implementiert man eine semantische Suche im Gewand eines Chatbots?

Vorbereitung: Wandle die vorhandene HTML-FAQ um in JSON (**beautifulsoup**). Tokenisiere die Fragen und wandle sie in Vektoren um mit einem LLM (jede Frage ein Vektor, Weg 2 aus dem [Kapitel „LLMs benutzen“](#)).

Anwendung: Tokenisiere den Nutzerinput und wandle ihn in einen Vektor um mit demselben LLM. Vergleiche mit Kosinus-Ähnlichkeit den Inputvektor zu den Fragenvektoren aus der Datenbank, zeige die Top-3-Treffer an. Zeige zum Top-Treffer die Antwort aus der Datenbank.

Technisch: Die Inferenz läuft vollständig im Browser (JavaScript, ONNX), ein winziges LLM genügt (20 MB; die Downloadzeit ist die Wartezeit, bis die Anwendung reagiert). Kein Server, der Fragen sieht, also kein Logging-Problem.

Beispiel: E-Mails klassifizieren

Aufgabe: Gegeben ein Postfach mit E-Mails der vergangenen Jahre, berechne für neu eingehende E-Mails, mit welcher Wahrscheinlichkeit sie beantwortet werden (= Score, ob eine E-Mail vermutlich eine Antwort erfordert).

- IMAP zum Download von Mails + Metadaten.
- Muss in „Sent“ die „Antwort“ zu einer Mail zuordnen (über **In-Reply-To-Header**) – das erzeugt die Labels.
- Ggf. Teildatensatz ausschließen (Spam o.ä.).
- Text der Mail in Vektor(en) umwandeln (LLM-Embedding).
- Vektoren in ein MLP zur Klassifikation füttern, supervised, mit BCE-Loss.

Beispiel: Empfehlungssystem

Aufgabe: Gegeben eine Liste besuchter bzw. gekaufter Items (Artikel, Produkte) pro Nutzer, empfehle einem neuen Nutzer ein Item, welches interessant ist (mit hoher Wahrscheinlichkeit angeklickt oder gekauft wird).

Lösungsansatz: Ein neuronales Netz konvertiert Items in Vektoren, sodass Items, die auf vielen Listen gemeinsam auftreten, zu Vektoren zugeordnet werden, die einen kleinen (Kosinus-)Abstand zueinander haben (Feature Engineering / Representation Learning). Dann finde im Latent Space Items in der Nähe vergangener Items eines Nutzers.

Wie trainieren wir das neuronale Netz? Die Listen der Nutzer sind „Sätze“, die Items sind „Wörter“ – das ist word2vec (Skip-gram) auf Kauflisten, bzw. node2vec auf dem bipartiten Nutzer-Item-Graphen aus dem [Kapitel über Representation Learning und Graphen](#). Alternativ bettet man Nutzer *und* Items in denselben Latent Space ein und trainiert supervised darauf, dass gekaufte Paare einen kleinen Abstand haben (*two-tower model*).

5.5.2 Vibecoding Deep Learning

Wir schauen uns praktisch an, wie man mit generiertem Code arbeiten kann. Beispiel-Aufgabe (das Empfehlungssystem von oben): Gegeben eine Liste besuchter bzw. gekaufter Items (Artikel, Produkte) pro Nutzer, empfehle einem neuen Nutzer ein Item, welches interessant ist (mit hoher Wahrscheinlichkeit angeklickt oder gekauft wird). Die Kompetenz, die dabei geübt wird, ist dieselbe, die auch die Übungsblätter verlangen: generierten Deep-Learning-Code anhand der Dokumentation kritisch prüfen – stimmt die Loss-Funktion, stimmen die Shapes, wird tatsächlich das trainiert, was behauptet wird?

5.5.3 Rezept: ein Anwendungskonzept schreiben

Die drei Beispiele oben folgen alle demselben Muster, und dasselbe Muster ist in der Klausur der Aufgabentyp „Anwendungskonzept“: Zu einer Aufgabenstellung aus der Praxis (Müll auf Fotos erkennen, Lagerbestände vorhersagen, Produktbeschreibungen erzeugen) soll man skizzieren, wie man sie mit Deep Learning angeht. Was dabei immer beantwortet werden muss, in dieser Reihenfolge:

1. **Was haben wir?** Welche Daten liegen vor, in welcher Modalität (Bild, Text, Tabelle, Zeitreihe, Graph), mit welchen Labels, wie viele. Was fehlt, und was können wir uns beschaffen oder selbst labeln.
2. **Was soll rauskommen?** Klassifikation, Regression, Ranking, Generierung oder nur eine Repräsentation für die Nachbarsuche. Damit steht die Ausgangsrichtung fest: Sigmoid, Softmax, linear oder gar keine.
3. **Welches Modell, und woher?** Fast nie selbst von null trainieren: ein vortrainiertes Modell als Feature-Extraktor einfrieren und einen kleinen Kopf trainieren oder finetunen oder ein fertiges Modell nur benutzen. Erst wenn das nicht reicht, eigene Architektur.
4. **Welcher Loss?** Er folgt aus Punkt 2: BCE für binär, Kreuzentropie für Klassen, MSE für Zahlen, Rekonstruktion für Autoencoder, kontrastiv für Paare. Wer zwei Ziele hat, addiert zwei Terme.
5. **Womit messen wir Güte?** Train-Test-Split, und eine Metrik, die zur Aufgabe passt: Accuracy nur bei balancierten Klassen, sonst Precision und Recall; bei Rankings Top-k; bei Generierung ein Mensch. Ohne diesen Punkt weiß man nicht, ob es funktioniert.
6. **Was kann schiefgehen?** Datenschutz und Logging, Kosten und Latenz, Verteilungsverschiebung zwischen Trainingsdaten und Einsatz, Overfitting auf das Testset, Bias in den Labels.

Wer diese sechs Punkte zu einer Aufgabe in je zwei Sätzen beantworten kann, hat ein Anwendungskonzept. Die Punkte 1 und 5 sind die, die in der Praxis am häufigsten vergessen werden.

5.5.4 Übung

Ein Verlag möchte eingehende Leserbriefe automatisch an die richtige Redaktion leiten (acht Ressorts). Gehen Sie die sechs Punkte des Rezepts durch: Was haben wir, was soll herauskommen, welches Modell und woher, welcher Loss, womit messen wir Güte, was kann schiefgehen. *Tipp: Die Antwort auf „welches Modell und woher“ ist in diesem Kapitel nie „selbst trainieren“, solange es ein vortrainiertes gibt.*

Antwortprobe: Beim Loss sollte etwas mit „Kreuzentropie“ dastehen, bei der Güte etwas anderes als „Accuracy“ – die Ressorts sind ungleich häufig.

5.6 Destillation

5.6.1 Was lernen wir beim Deep Learning eigentlich?

Ansatz 1: Regression zu bekannter Funktion

Wenn eine Funktion der Daten bekannt ist, also $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ für Datenpunkte $x_1, \dots, x_N \in \mathbb{R}^n$ bekannt ($y_i = f(x_i)$ liegen vor) oder billig zu berechnen ist, können wir ein neuronales Netz $\mathcal{N}$ trainieren, sodass $\mathcal{N}(x_i) \approx y_i$ ist (Regression, MSE-Loss). Analog für $c: \mathbb{R}^n \rightarrow \{0, 1\}$ eine Klassifikation: Wenn $y_i = c(x_i)$ bekannt sind, lässt sich $\mathcal{N}$ trainieren, sodass $\mathcal{N}(x_i) = y_i$ in möglichst vielen Fällen (Klassifikation, BCE-Loss).

Problem: Manchmal kennen wir die Funktion f bzw. c nicht – z.B. wenn wir Daten ohne Label haben (aber klassifizieren oder clustern wollen). Oder wenn die gesuchte Funktion f komplizierter ist, z.B. wenn die Eingabe x eine Sequenz von Token $x_1, \dots, x_k$ repräsentiert und

$$f(x_1, \dots, x_k) := P(x_k \mid x_1, \dots, x_{k-1}), \quad (137)$$

die Wahrscheinlichkeit, eine Sequenz $x_1, \dots, x_k$ vorzufinden, wenn man $x_1, \dots, x_{k-1}$ gesehen hat (*next token prediction*). Das kann man immer noch aus den Daten schätzen, aber relativ teuer.

Ansatz 2: Regression zu unbekannter Funktion

Beispiel Autoencoder: Wir trainieren eine Regression des neuronalen Netzes $\mathcal{N}$ auf die Identitätsfunktion (wie in Ansatz 1), aber $\mathcal{N} = D \circ E$ (Decoder nach Encoder mit Bottleneck dazwischen). Die Abbildung $E: \mathbb{R}^n \rightarrow \mathbb{R}^m$, die der Encoder lernt, ist vor dem Training völlig unbekannt. Es gibt mehr als eine „korrekte“ solche Abbildung. Die Menge aller möglichen Funktionen $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ ist sehr groß; welche Teilmenge zu sinnvollen Encodern gehört, ist a priori unklar.

5.6.2 Model Compression

Zwei Ansätze vereint: Suchproblem und Regression

Die *Suche* nach der richtigen Funktion braucht viel Platz und Flexibilität, daher sind große (tiefe) neuronale Netze besser geeignet zum Lernen unbekannter Funktionen. Eine *Regression* auf eine bekannte Funktion braucht nicht viel Flexibilität, daher lassen sich auch flache neuronale Netze einsetzen; die benötigte Komplexität hängt nur von der Funktion ab, nicht davon, wie schwer es ist, diese zu erraten.

Das legt nahe: erst mit einem großen Netz suchen, dann das Ergebnis in ein kleines Netz überführen. Zwei Strategien:

Strategie 1: Pruning. Beim Pruning (Abschneiden) werden Bereiche des neuronalen Netzes identifiziert, die wenig zur Berechnung beitragen, und entfernt (Gewichte auf 0 gesetzt, Layer entfernt). Es ist schwierig auszuwählen, was weg kann, und nach dem Entfernen „unwichtiger“ Teile ist oft ein Finetuning erforderlich.

Strategie 2: Destillation. Beim Destillieren wird ein kleineres **Schüler**-Netzwerk trainiert mit Hilfe eines größeren **Lehrer**-Netzwerks (welches zuvor teuer trainiert wurde). Dabei gibt es 3 wesentliche Ansätze:

1. Lehrer-Labels auf Daten ermöglichen supervised Training der Schüler. (Lehrer-Training ist „Suche“, Schüler-Training ist „Regression“.)
2. Synthetische Daten, generiert durch das Lehrer-Netzwerk, beschleunigen das Schüler-Training.
3. Der Schüler lernt direkt aus den Logits das darin verborgene Wissen. (Destillation im engeren Sinne.)

Vorläufer: Ensemble Model Compression

Bucilua, Caruana, Niculescu-Mizil: *Model Compression* (2006):

1. Trainiere ein großes Ensemble-Modell (z.B. Random Forest, XGBoost) auf den Daten,
2. generiere synthetische Eingabedaten,
3. generiere Labels mit dem trainierten Ensemble-Modell,
4. trainiere ein neuronales Netz auf den neuen, viel umfangreicheren Daten.

Das Training wirkt regularisierend, das neuronale Netz verallgemeinert besser – und ist zur Inferenz um Größenordnungen schneller als das Ensemble.

Wie deep muss es sein?

Ba, Caruana: *Do Deep Nets Really Need to be Deep?* (2013). Wie muss ein neuronales Netz aufgestellt sein (Architektur), um eine Funktion zu lernen? Zwei getrennte Fragen:

1. **Expressivity:** Ist das Netz komplex genug, um die Funktion abzubilden?
2. **Learnability:** Kann der Gradientenabstieg diese Funktion im großen Suchraum finden?

Ergebnisse: Expressivity ist nicht das Problem (vgl. das [Kapitel „Universelle Approximation“](#)), flache und breite Netzwerke sind stark. Learnability ist das Problem flacher Netze: Ein flaches Netz, das mit den Soft-Labels eines tiefen Netzes trainiert wird, erreicht dessen Genauigkeit – dasselbe flache Netz direkt auf den Daten trainiert aber nicht.

5.6.3 Model Distillation

Distillation Loss

Hinton, Vinyals, Dean: *Distilling the Knowledge in a Neural Network* (2015). Für eine k -Klassen-Klassifikationsaufgabe ist der letzte Schritt im großen Lehrer-Modell ein Softmax $\mathbb{R}^k \rightarrow [0, 1]^k$, gefolgt von der „harten“ Klassifikation argmax , um zu einem Klassenurteil zu kommen. Idee: Lerne nicht aus dem harten, sondern aus dem *weichen* Urteil. Und mach es noch weicher.

Loss-Funktion zum Schüler-Training: KL-Divergenz (relative Entropie) zwischen Schüler-Verteilung und Lehrer-Verteilung, jeweils reskaliert mit Temperatur T :

$$q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}, \quad L_{\text{distill}} = T^2 \cdot D_{KL}(q^{\text{Lehrer}} \| q^{\text{Schüler}}). \quad (138)$$

Mit $T > 1$ werden die kleinen Wahrscheinlichkeiten der falschen Klassen sichtbar (Temperature Scaling im [Kapitel „Latente Räume geometrisch“](#)), und genau die tragen die Information. In der Praxis kombiniert man das mit dem gewöhnlichen Kreuzentropie-Loss auf den echten Labels bei $T = 1$.

Dark Knowledge

Beispiel: fehlerhaft gelabelte Daten, z.B. ein Hund x mit dem Label „Katze“. Der Lehrer klassifiziert nach dem Lernen „Katze“ (leider falsch). Aber vor dem argmax sehen wir vermutlich $P(x = \text{Hund}) > 0$, und zwar deutlich – d.h. in $P(x = \text{Hund})$ steckt noch Information, „dunkles Wissen“: Der Lehrer weiß, dass das Bild einem Hund ähnlich sieht, auch wenn er es der falschen Klasse zuordnet. Der Schüler lernt aus dunklem Wissen und leidet weniger unter falschen Labels. Allgemeiner: Die Ähnlichkeitsstruktur zwischen den Klassen (eine 7 sieht eher wie eine 1 aus als wie eine 0) wird mitübertragen.

Schüler übertreffen Lehrer

Furlanello, Lipton, Tschannen, Itti, Anandkumar: *Born Again Neural Networks* (2018), inspiriert von „born again trees“ aus 1996. Das Schüler-Netzwerk ist nicht nur kleiner und damit schneller und billiger als das Lehrer-Netzwerk – es kann sogar *besser* werden, selbst bei gleicher Architektur. Die implizite Neugewichtung der Trainingsdaten führt zu Fokus auf „schwere Fälle“; der Loss berücksichtigt weniger den einfachen Teil der Aufgabe. Knowledge Distillation ist damit Regularisierung bzw. Optimierung statt nur Kompression. Und man kann iterieren: ein Ensemble aus Generationen von Schülern.

DistilBERT

Sanh, Debut, Chaumond, Wolf: *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter* (2019): „we leverage knowledge distillation during the pre-training phase and show that it is possible to reduce the size of a BERT model by 40 %, while retaining 97 % of its language understanding capabilities and being 60 % faster. To leverage the inductive biases learned by larger models during pre-training, we introduce a triple loss combining language modeling, distillation and cosine-distance losses.“

DistilBERT := jeder zweite Layer von BERT (als Initialisierung). Der Loss ist die Summe aus drei Teilen:

- **Distillation Loss:** KL-Divergenz zu den temperatur-skalierten Logits des Lehrers.
- **Masked Language Modeling Loss:** wie BERT, Lernen aus den Daten, mit Temperatur 1.

- **Cosine-Distance Loss:** latente Mannigfaltigkeiten beibehalten, d.h. die Outputs der Layer (die hidden states) von Schüler und Lehrer im Kosinusabstand vergleichen.

Das ist das Modell aus dem [Kapitel „Vortrainierte Modelle“](#) für Text Classification und Fill-Mask.

5.6.4 Anwendungen

Speculative Decoding

Vgl. Yan, Agarwal, Venkataraman: *Decoding Speculative Decoding* (2024). Problem: Das (autoregressive) Sampling eines nächsten Tokens mit einem LLM ist teuer, benötigt aber ein Ergebnis, um das darauf folgende Token zu sampeln – man kann nicht parallelisieren. Trick: Rate die kommenden 5 Token mit einem kleinen, billigen Modell, und *verifiziere* sie mit dem teuren LLM in einem einzigen Forward-Pass (der ist parallel). Stimmen sie, hat man 5 Token zum Preis von einem; sonst nimmt man das erste abweichende Token des großen Modells und rät neu. Trick 2: Das billige Modell ist vom LLM destilliert, damit es möglichst oft richtig rät.

SDXL Turbo

Sauer, Lorenz, Blattmann, Rombach: *Adversarial Diffusion Distillation* (2023). Aus einem vortrainierten tiefen Diffusionsmodell (SDXL) zum Generieren von Bildern aus Bildbeschreibungen in einem k -Schritte-Iterationsprozess des Entrauschens wird ein 1-Schritt-Modell destilliert – mit Distillation Loss (Schüler-Output soll dem Lehrer-Output nach k Schritten ähneln) und Adversarial Loss (GAN-Training aus dem [Kapitel „GANs“](#), damit die Bilder scharf bleiben). Bildgenerierung in Echtzeit.

Gemini Flash

Google Gemini Team: *Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context* (2024). Auch Google benutzt Destillation, um aus dem großen, teuren Gemini 1.5 Pro das günstigere, schnellere Gemini 1.5 Flash zu destillieren. Analog: OpenAIs GPT-Mini-Modelle.

Chain of Thought

1. Bringe ein LLM dazu, länger bzw. mehr nachzudenken (= mehr Output zu produzieren): die „Chain of Thought“ – warum diese Antwort, Begründungen, Zwischenschritte.
2. Verstecke langwierige oder schlechte Zwischenüberlegungen und lass Schüler aus den „wohlüberlegten“ Inhalten lernen, mit Destillation.

So entstehen „Reasoning“-Modelle, die kompakt das können, wofür der Lehrer viele Token gebraucht hat.

5.6.5 Übung

Ein Lehrer-Modell gibt für drei Klassen die Logits $(3, 1, 0)$ aus.

1. Berechnen Sie die Softmax-Ausgabe bei $T = 1$ und bei $T = 2$ (also mit z_j/T im Exponenten). *Tipp:* $e^3 \approx 20,1$, $e^{1,5} \approx 4,48$, $e \approx 2,72$, $e^{0,5} \approx 1,65$.
2. Welche Information steckt in der Ausgabe bei $T = 2$, die bei $T = 1$ fast verschwunden ist? Wie heißt sie im Kapitel?
3. Warum wird der Distillation Loss mit T^2 multipliziert?

Antwortprobe: Bei beiden Temperaturen summieren sich die drei Werte zu 1, aber der größte ist bei $T = 2$ deutlich kleiner.

5.7 Optimierer: Momentum, Adam, Muon

5.7.1 Adam oder was?

Wir haben die Grundidee des stochastischen Mini-Batch-Gradientenabstiegs gelernt und dann gesagt: In PyTorch klappt Adam oft besser als SGD. Aber was steckt dahinter?

Das Grundproblem beim SGD

Die Daten $x_1, \dots, x_N$ modellieren wir als Stichprobe einer Zufallsvariable X . Die Verteilung von X ist unbekannt, und die Stichprobe hat Sampling-Bias. Am liebsten möchten wir $N \rightarrow \infty$ haben, damit aus der Stichprobe die Verteilung von X genau geschätzt werden kann. Für eine Stichprobe können wir den Loss ausrechnen; idealisiert für eine „unendlich große Stichprobe“ wäre das der Erwartungswert der Loss-Funktion unter der Verteilung von X – der „eigentlich richtige Loss“. Es gibt also drei Player:

- Der Erwartungswert $\mathbb{E}_X[L(X; w)]$ der Loss-Funktion unter der Verteilung von X (unbekannt).
- Ein Schätzwert für diesen Erwartungswert, berechnet aus einer großen Stichprobe: $\frac{1}{N} \sum_i L(x_i; w)$. Bester Schätzwert, wenn Stichprobe = alle Daten, die man kriegen kann.
- Ein schlechter, aber billiger Schätzwert, berechnet aus einem Mini-Batch B : $\frac{1}{|B|} \sum_{i \in B} L(x_i; w)$.

Pro Mini-Batch wird der Loss grob abgeschätzt und damit werden Gradienten berechnet, die also auch „nur so grob geschätzt“ sind.

Verständnisfrage

(Warum) kann der Loss von einem Mini-Batch zum nächsten *steigen*, wenn wir stochastischen Gradientenabstieg verwenden? Haben wir nicht gelernt, dass Gradientenabstieg den Loss senken muss?

Idee: Momentum

Wir schätzen den „wahren Loss“ und damit den eigentlich interessanten Gradienten aus jedem Mini-Batch neu. Die Schätzung wird besser, wenn die Batch Size steigt. Kann man auch Informationen aus dem vorherigen Mini-Batch nutzen, um einen ähnlichen Effekt zu erzielen? Man könnte z.B. die Gradientenschätzungen von je 5 Batches mitteln und erhielte damit eine bessere Schätzung. Das praktische Problem mit diesem Ansatz: Man müsste 5 Gradienten (jeder so groß wie das Modell) speichern, und man bekäme nur alle 5 Batches ein Update.

Idee: Gradienten sind ein bisschen ähnlich, wenn man sich nicht so weit weg bewegt im Loss Landscape. Statt des Mittelwerts der letzten 5 Gradienten verwende ein *exponentiell gewichtetes* Mittel, das sich mit einem einzigen Vektor mitführen lässt:

$$m_t := \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad w_{t+1} := w_t - \rho m_t, \quad (139)$$

mit g_t dem Gradienten des aktuellen Batches und typischerweise $\beta_1 = 0,9$. Visuelle Metapher: Zeigt der Pfeil (Gradient) in eine Richtung, geht es nächstes Mal auf jeden Fall auch ein bisschen weiter in diese Richtung – Impulserhaltung, daher **Momentum**. Rauschen in wechselnden Richtungen mittelt sich heraus, konsistente Richtungen verstärken sich.

Idee: Adaptive Lernrate

Der zweite Bestandteil von **Adam** (= Adaptive Moment Estimation) ist der adaptive Teil: Zusätzlich zu einer Heuristik zur besseren Schätzung der Gradienten der Loss-Funktion berechne die Quadrate – das mittlere Gradientenquadrat, also das *zweite Moment* (nicht die Varianz; dafür müsste man noch m_t^2 abziehen), ebenfalls als exponentiell gewichtetes Mittel, komponentenweise:

$$v_t := \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \quad w_{t+1} := w_t - \rho \frac{m_t}{\sqrt{v_t} + \epsilon}, \quad (140)$$

mit typischerweise $\beta_2 = 0,999$ und $\epsilon = 10^{-8}$ gegen Division durch null. Sind die Gradienten einer Koordinate durchweg groß, wird der Schritt dort gedämpft; sind sie durchweg klein, wird er verstärkt. Im Ergebnis ist $m_t / \sqrt{v_t}$ von der Größenordnung 1, jeder Parameter macht also Schritte der Größenordnung ρ – unabhängig davon, wie groß seine Gradienten absolut sind. Diese Technik kommt von RMSProp (Root Mean Square Propagation). Der Effekt ist eine **Lernrate pro Gewicht**: Gewichte mit selten großen Gradienten (z.B. Embeddings seltener Wörter) bekommen dadurch relativ größere Schritte.

Adam oder was?

Dahinter steckt also eine adaptive Lernrate, abhängig von der bisherigen Größe der Gradienten, einzeln für jedes Gewicht, sowie die Impuls-Heuristik zur verbesserten Gradientenschätzung. (Adam korrigiert außerdem die Schätzungen m_t , v_t in den ersten Schritten für den Bias durch die Initialisierung mit null.) Wo geht das schief? Wenn die Lernrate nicht stark genug abgesenkt wird, findet die Optimierung nie das lokale Optimum und oszilliert darum herum. Deshalb kombiniert man Adam in der Praxis mit einem *Learning Rate Schedule*, der ρ im Laufe des Trainings verkleinert.

Adam oder Muon?

Die Grundidee „wir rechnen clever eine bessere Schätzung bzw. Heuristik für Gradienten aus“ kann man noch weiter treiben. Idee: Ein linearer Layer im neuronalen Netz lässt sich als *Operator* betrachten, also als Abbildung (die Matrixmultiplikation). Behandle die Gewichte einer Matrix nicht isoliert, sondern als Ganzes! Und statt mit der Varianz zu reskalieren wie bei Adam, **orthogonalisiere** das Update – das ist etwas schwieriger zu verstehen.

Erinnerung: Eine Matrix A hat eine Singulärwertzerlegung $A = UDV$, wobei U und V orthogonale Matrizen sind (bestimmte Koordinatensysteme) und D eine Diagonalmatrix mit den Singulärwerten. Wenn die Einträge von D entweder 0 oder 1 sind, ist A selbst (semi-)orthogonal.

Muon: Wir berechnen erst wie bei SGD den Gradienten der Loss-Funktion nach einer Gewichtsmatrix, hier genannt A ; die Update-Formel wäre $W_{\text{neu}} := W_{\text{alt}} - \rho A$. „Orthogonal machen“ durch $A = UDV$ berechnen und $A_{\text{ortho}} := UD_{\text{ortho}}V$ mit $D_{\text{ortho},i} := [D_i > 0]$ ist leider teuer: Die SVD ist aufwändig. Also mit der Brechstange:

- Wenn $A = UDV$ die Singulärwertzerlegung ist, dann ist $f(A) = Uf(D)V$ für ein Polynom f (in A und $A^T A$).
- Approximiere die Funktion $[x > 0]$ auf $(0, 1]$ mit einem Polynom f kleinen Grades (Newton-Schulz-Iteration).
- Oha: $g(x) := f(f(f(f(f(x)))))$ approximiert $[x > 0]$ auch ganz gut (*NewtonSchulz5*).
- Berechne $A_{\text{oha}} := g(A)$ – nur Matrixmultiplikationen, die auf der GPU billig sind – und hoffe, dass es nah dran ist an A_{ortho} .
- Update $W_{\text{neu}} := W_{\text{alt}} - \rho A_{\text{oha}}$.

Jordan, Keller et al.: *Muon: An optimizer for hidden layers in neural networks* (2024). Alle Richtungen im Gewichtsraum bekommen dadurch denselben Schritt, nicht nur die mit großem Singulärwert – Richtungen, die der Gradient „vernachlässigt“, werden mitgezogen.

Wieso so kompliziert?

Genau genommen liegt der Gradient der Loss-Funktion an einem Punkt des Gewichtsraums im *Kotangentenraum* des Graphen der Loss-Funktion – er ist eine Linearform, die Änderungsrichtungen Zahlen zuordnet ([Kapitel „Tensoren“](#): Kovektoren!). Um einen Vektor aus dem Gewichtsraum daraus zu machen, mit dem wir einen Schritt gehen können, müssen wir „dualisieren“. Dabei kann man verschiedene Geometrien betrachten:

- Die **euklidische Geometrie**, einfach $\mathbb{R}^n$ mit dem Standardskalarprodukt; dann kann man die Zahlen, die man ausrechnet, wie gehabt als Vektor in $\mathbb{R}^n$ auffassen. Das ist SGD.
- Die **Riemannsche Geometrie**, die durch die Fisher-Informationsmatrix gegeben ist – der *Natural Gradient* (Amari). Die Fisher-Matrix ist die zweite Ableitung der relativen Entropie; Schritte werden so gemessen, wie stark sich die *Ausgabeverteilung* des Modells ändert, nicht die Gewichte.
- Verschiedene andere sogenannte affine Geometrien; das ist ein aktives Forschungsgebiet in der Optimierungstheorie, Stichwort **Informationsgeometrie**. Z.B. Derun Zhou, Keisuke Yano, Mahito Sugiyama: *Dual Riemannian Newton Method on Statistical Manifolds* (2025), [arXiv:2511.11318](#).

Adam und Muon sind in diesem Bild billige Näherungen an eine bessere Geometrie des Gewichtsraums.

5.7.2 Übung

Eine Funktion mit Gradienten $g_1 = 1$, $g_2 = 1$, $g_3 = 1$ in drei aufeinanderfolgenden Schritten (ein langes, flaches Tal), Lernrate $\rho = 0,1$.

1. Wie weit ist SGD nach drei Schritten gekommen?
2. Wie weit kommt Momentum mit $v_t = \beta v_{t-1} + g_t$, $\beta = 0,9$ und demselben ρ ? Das ist die Form, die `torch.optim.SGD(momentum=0.9)` benutzt; sie unterscheidet sich von der Fassung oben im Kapitel um den Faktor $(1 - \beta)$, der dort das Mittel normiert. Mit der Kapitelfassung käme entsprechend ein Zehntel heraus. *Tipp: $v_1 = 1$, $v_2 = 1,9$, $v_3 = 2,71$ – Sie müssen nur noch aufsummieren.*
3. Jetzt wechselt der Gradient das Vorzeichen: $g = (1, -1, 1)$. Was macht Momentum, und warum ist das in einer engen Schlucht erwünscht?

Antwortprobe: Bei (2) ist der zurückgelegte Weg knapp doppelt so lang wie bei (1).

Weiterlesen

Prince, *UDL*, Abschnitte 6.3 und 6.4 (Momentum, Adam); Deisenroth et al., *MML*, Kap. 7 (Continuous optimization). Beide Bücher sind frei verfügbar, siehe [Über dieses Skript](#).

5.8 Adversarial Attacks und Polysemie

Zwei Themen zum Abschluss, die über die Vorlesung hinausgehen, aber mit dem Gelernten schon verständlich sind.

5.8.1 Adversarial Attacks

Definition

Die gezielte Manipulation des Inputs eines Machine-Learning-Modells, um ein spezifisch falsches Ergebnis oder ein unerwartetes Verhalten zu produzieren. Die Manipulation ist häufig nicht direkt für Menschen zu erkennen.

Beispiel: die gezielte Manipulation des Inputs eines Bildklassifikators, um eine spezifisch falsche Klasse zu produzieren. Also: Wir stecken einen weißen Hai rein, aber „die KI“ sagt „Goldfisch“ dazu. Reingelegt!

Theorie

Wir betrachten den Eingaberaum „Menge aller Bilddateien“ $([0,1]^{3 \cdot H \cdot W})$ sowie die Eingabemenge „Menge aller Trainingsbilder“. Das Modell lernt Entscheidungsgrenzen im Eingaberaum, um Klassenlabel zu vergeben. Hypothese: Nahe der Entscheidungsgrenze liegen *adversarial examples* – und da die Trainingsbilder nur auf einer winzigen Teilmenge des Eingaberaums liegen (Manifold-Hypothese), verläuft die Entscheidungsgrenze fast überall dicht an jedem Bild vorbei, nur in Richtungen, die „aus der Mannigfaltigkeit heraus“ zeigen und für Menschen wie Rauschen aussehen.

Das ist die These der Bachelorarbeit „Das Dimpled Manifold Model als Erklärung für die Existenz und die Eigenschaften von Adversarial Examples“ von N. Wiczorek (HHU 2023), nach Shamir et al. 2021: Die Entscheidungsgrenze schmiegt sich wie eine Folie mit kleinen Dellen (*dimples*) an die Datenmannigfaltigkeit an.

Wie wird's gemacht?

Mit Gradientenabstiegsmethoden sucht man, ausgehend von einem Bild eines weißen Hais, Richtungen, die die Klassifikation ändern, ohne besonders viel am Bild zu ändern: ein Schritt in Richtung des Gradienten des Loss nach dem *Bild* oder viele kleine Schritte. Anders als sonst: Gradientenabstieg im **Eingaberaum**, nicht im Gewichtsraum – Backpropagation liefert ∇_x genauso wie ∇_w , denn im DAG ist x nur ein weiteres Blatt. Genau das tut auch das „Dreaming“ im Notebook `latent-classifier.ipynb` zum [Kapitel „LLMs benutzen“](#): Es sucht per Adam im Raum der Eingabe-Embeddings nach einem Text, der den Spam-Detektor maximal triggert – ein adversarial example ohne den Anspruch, unauffällig zu sein.

Vertiefung: Fast Gradient Sign Method

Goodfellow, Shlens und Szegedy 2014: Mit dem Loss $L(f(x), y)$ für das wahre Label y ist

$$x_{\text{adv}} := x + \epsilon \cdot \text{sign}(\nabla_x L(f(x), y)) \quad (141)$$

ein Bild, das sich in jedem Pixel um höchstens ϵ vom Original unterscheidet und den Loss in erster Näherung maximal vergrößert. Gezielt auf eine Wunschklasse y_{Ziel} : Minus statt Plus und y_{Ziel} statt y . Ein einziger Gradientenschritt genügt oft, weil das Modell in der Umgebung des Bildes fast linear ist; die iterative Variante (PGD) macht viele kleine Schritte und projiziert nach jedem zurück in den ϵ -Würfel um x .

Dass das nicht nur mit Pixeln funktioniert: Sharif et al. 2016, *Accessorize to a Crime: Real and Stealthy Attacks on State-of-the-Art Face Recognition* – eine bedruckte Brillenfassung lässt ein Gesichtserkennungssystem eine andere Person erkennen. Und aus dem Heise-Newsticker vom 2.2.2026: [Manipulierte Straßenschilder: Wenn das autonome Auto Fußgänger umfährt.](#)

5.8.2 Polysemie und die Manifold-Hypothese

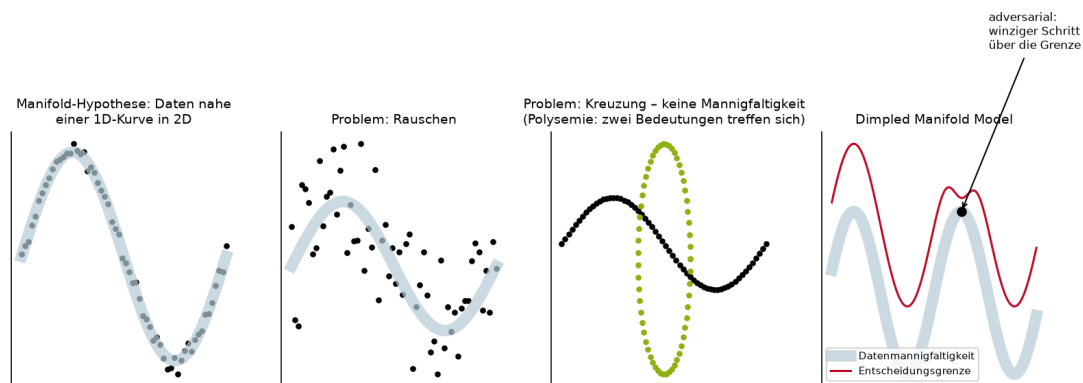
(Folien von Ben Ruppik, aus der Vorlesung *Topological Data Analysis* im Master AI & Data Science, Juni 2025.)

Manifold Hypothesis: High-dimensional data lies on or near a low-dimensional structure called a *manifold*. Das ist die Annahme hinter allem, was wir über latente Räume gesagt haben: Der Autoencoder

findet die Parametrisierung der Mannigfaltigkeit, der VAE glättet sie, word2vec legt Wörter auf eine Mannigfaltigkeit, auf der Richtungen Bedeutung haben.

Problems with the Manifold Hypothesis: Sometimes data does *not* have a low-dimensional structure and cannot be effectively mapped onto a lower-dimensional space – wegen Rauschen oder wegen zu wenigen Samples, sodass die zugrundeliegende Struktur nicht geschätzt werden kann. Und manchmal ist die Struktur zwar niedrigdimensional, aber keine Mannigfaltigkeit: Zwei Kurven, die sich kreuzen, haben am Schnittpunkt keine eindeutige Tangente.

Genau das passiert bei **Polysemie**: Ein Wort wie „Maus“ (Tier, Computermouse) oder „Bank“ (Sitzgelegenheit, Geldinstitut) hat mehrere Bedeutungen, die in verschiedenen Kontexten vorkommen. Im Embedding-Raum eines kontextabhängigen Modells (BERT) liegen die Vorkommen von „Maus“ auf mehreren „Ästen“, die sich an einem Punkt treffen – eine *Singularität* der Datenmannigfaltigkeit. Topologische Datenanalyse (persistent homology, lokale Dimension) kann solche Singularitäten finden und damit polyseme Wörter *ohne Wörterbuch* aus den Embeddings entdecken. word2vec mit einem Vektor pro Wort kann das prinzipiell nicht: Es muss alle Bedeutungen in einen Punkt mitteln.



Die Manifold-Hypothese und ihre Grenzen: Daten nahe einer Kurve; Rauschen; zwei sich kreuzende Kurven (keine Mannigfaltigkeit – so sieht Polysemie im Embedding-Raum aus); und das Dimpled Manifold Model, in dem die Entscheidungsgrenze sich an die Daten schmiegt und Adversarial Examples nur einen winzigen Schritt entfernt sind.

Damit schließt sich der Kreis zur Kernfrage aus dem [Kapitel „Latente Räume geometrisch“](#): Welche Struktur finden wir im Latent Space? Manchmal eine schöne Mannigfaltigkeit – und manchmal etwas Interessanteres.

5.8.3 Übung

Ein Bildausschnitt $x = (0,50, 0,20, 0,90)$, der Gradient des Loss nach x sei $(0,03, -0,07, 0,01)$, und $\varepsilon = 0,02$.

1. Berechnen Sie $x' = x + \varepsilon \cdot \text{sign}(\nabla_x L)$.
2. Wie groß ist die größte Änderung eines einzelnen Pixels? Wie groß ist sie in 8-Bit-Stufen (also mal 255)? *Tipp: Genau deshalb sieht man die Störung nicht.*
3. Warum benutzt FGSM nur das **Vorzeichen** des Gradienten und nicht seinen Betrag?

Antwortprobe: Alle drei Pixel ändern sich um denselben Betrag.

6 Anhang

6.1 Glossar zur Klausurvorbereitung

Die Begriffe, die in Klausuraufgaben vorkommen, je in ein bis drei Sätzen, mit Link zum Kapitel. Das Glossar ersetzt nicht die Kapitel; es ist zum Abhaken gedacht: Wer einen Eintrag liest und nicht erklären kann, warum er stimmt, liest das Kapitel.

6.1.1 Grundbegriffe

Deep Learning ist das Training künstlicher neuronaler Netze mit vielen Schichten. Es ist ein Teilgebiet des maschinellen Lernens und orthogonal zur Frage supervised, unsupervised oder reinforcement. [Was ist Deep Learning?](#)

Training und Inferenz. Training: Daten rein, gefittetes Modell (Gewichte) raus. Inferenz: neue Daten und Gewichte rein, Vorhersage raus. [Was ist Deep Learning?](#)

Train-Test-Split, Validation-Split. Testdaten werden zurückgehalten, um die Güte zu messen. Wer Hyperparameter nach dem Testset wählt, overfittet indirekt darauf; dafür spaltet man vom Trainingsset ein Validation-Set ab. [Was ist Deep Learning?](#)

Accuracy, Precision, Recall. Accuracy ist der Anteil richtiger Vorhersagen und bei unbalancierten Klassen irreführend. Precision: Anteil der als positiv vorhergesagten, die wirklich positiv sind. Recall: Anteil der wirklich positiven, die gefunden wurden. [Was ist Deep Learning?](#)

Over- und Underfitting. Overfitting: gut auf Trainings-, schlecht auf Testdaten, das Modell hat auswendig gelernt. Underfitting: schon auf Trainingsdaten schlecht. Gegenmittel gegen Overfitting sind Regularisierung, Early Stopping, Data Augmentation, Dropout. [Autoencoder](#)

Generatives Modell. Ein Modell für eine Wahrscheinlichkeitsverteilung, als Dichte oder als Sampler. Generieren heißt aus dieser Verteilung sampeln. [Was ist Deep Learning?](#)

6.1.2 Neuron, Perzeptron, Netz

Iverson-Klammer. $[A] = 1$, wenn A wahr ist, sonst 0. Das Perzeptron ist $f(x) = [w \cdot x + b > 0]$. [Neuron und Perzeptron](#)

Perzeptron ohne Bias kann nur Hyperebenen durch den Ursprung lernen. Der Bias lässt sich als nulltes Gewicht mit konstanter Eingabe 1 schreiben. [Neuron und Perzeptron](#)

Linear trennbar. Zwei Klassen, die eine Hyperebene trennt. Nur solche kann ein einzelnes Perzeptron lernen; XOR ist das Gegenbeispiel. [Neuron und Perzeptron](#)

Perzeptronregel und Konvergenz. $w_{\text{neu}} = w + \rho(y - f(x))x$. Bei linear trennbaren Daten findet die Regel nach endlich vielen Fehlern eine trennende Hyperebene; sonst terminiert sie nie. [Das Perzeptron trainieren](#)

Verallgemeinertes Perzeptron. Skalarprodukt, dann Aktivierungsfunktion τ : lineare Regression ($\tau = \text{id}$), Perzeptron (Sprung), logistische Regression ($\tau = \sigma$). [Das Perzeptron trainieren](#)

Warum Schichten ohne Nichtlinearität nichts bringen. Ein Produkt von Matrizen ist eine Matrix; gestapelte lineare Regressionen sind eine lineare Regression. [Das Perzeptron trainieren](#)

Tensor. Für uns ein mehrdimensionales Array. Die Zahl der Achsen heißt *Stufe* oder, englisch, *rank*: Rang 0 ist ein Skalar, Rang 1 ein Vektor, Rang 2 eine Matrix (nicht zu verwechseln mit dem Rang einer Matrix aus der linearen Algebra). Die erste Achse ist in PyTorch fast immer der Batch – außer bei `nn.RNN`, `nn.LSTM`, `nn.GRU` und `nn.Transformer`, die voreingestellt (`seq`, `batch`, `feature`) erwarten. `reshape` ändert nur die Klammerung, nicht die Daten. [Tensoren](#)

6.1.3 Logistische Regression und Loss

Odds und Logit. $\text{odds}(p) = p/(1 - p)$, bei $p = \frac{1}{2}$ also 1:1. $\text{logit}(p) = \ln \text{odds}(p)$, die Umkehrfunktion von σ . [Logistische Regression](#)

Sigmoid. $\sigma(x) = 1/(1 + e^{-x})$, Werte in $(0, 1)$, $\sigma(-x) = 1 - \sigma(x)$, $\sigma' = \sigma(1 - \sigma)$. Die Ausgabe ist eine Wahrscheinlichkeit, wenn die Eingabe ein Logit ist. [Logistische Regression](#)

Softmax. $\text{sm}_j(v) = e^{v_j} / \sum_k e^{v_k}$, ohne Minus im Exponenten; summiert zu 1, verallgemeinert σ auf K Klassen, nur Differenzen der Logits zählen. Für sich nicht ausschließende Klassen (Tags) stattdessen ein Sigmoid pro Klasse. [Logistische Regression](#)

Kreuzentropie, NLL, KL-Divergenz. Die Kreuzentropie ist die negative Log-Likelihood des Modells und unterscheidet sich von der KL-Divergenz nur um die Entropie der wahren Verteilung, die nicht von den Gewichten abhängt. [Loss-Funktionen und Maximum Likelihood](#)

Maximum Likelihood. Wähle die Gewichte, unter denen die Daten am wahrscheinlichsten sind. Liefert BCE für die logistische und MSE für die lineare Regression mit Gauß-Rauschen. [Loss-Funktionen und Maximum Likelihood](#)

Hinge Loss. $\max(0, 1 - f(x))$ für $y = 1$: verlangt einen Sicherheitsabstand, der Loss der Support Vector Machine. Der Perceptron Loss ist $\max(0, -f(x))$. [Gradientenabstieg](#)

6.1.4 Gradientenabstieg

Gradient und Update. Der Gradient zeigt bergauf; $w_{\text{neu}} = w_{\text{alt}} - \rho L'(w_{\text{alt}})$ geht bergab. Beim Ableiten immer sagen, wonach: nach w , die Daten sind Konstanten. [Gradientenabstieg](#)

Kettenregel. Die Ableitung einer Verkettung ist das Matrixprodukt der Jacobi-Matrizen. Geht eine Variable auf mehreren Pfaden ein, addieren sich die Beiträge. [Gradientenabstieg](#)

Lernrate. Zu klein: ewig langsam. Zu groß: springt über das Tal, Loss steigt oder wird nan. Hängt von der Skalierung der Features ab, deshalb standardisieren. [Gradientenabstieg](#)

SGD, Mini-Batch, Epoche. SGD schätzt den Gradienten aus einem Teil der Daten statt aus allen; der Schätzer ist verrauscht, aber im Mittel korrekt. Ein Mini-Batch sind einige Dutzend Samples, eine Epoche ein Durchlauf durch alle Daten. Der Loss darf von Batch zu Batch steigen. [Gradientenabstieg](#)

Sattelpunkt. Ein Punkt mit Gradient null, der in einer Richtung Minimum und in einer anderen Maximum ist. In hohen Dimensionen häufiger als lokale Minima; Momentum hilft, ihn zu verlassen. [Optimierer](#)

Momentum, Adam. Momentum mittelt Gradienten über Schritte, Adam passt zusätzlich die Lernrate pro Gewicht an die bisherige Größe der Gradienten an (mittleres Gradientenquadrat). Faustregel im Kurs: Adam statt SGD. [Optimierer](#)

6.1.5 Backpropagation und Autograd

Backpropagation. Kettenregel plus Buchführung: im Forward Pass Zwischenergebnisse merken, dann rückwärts Schicht für Schicht das Fehlersignal mit der transponierten Gewichtsmatrix und der Ableitung der Aktivierung weiterreichen. Aufwand etwa das Doppelte einer Inferenz. [Backpropagation](#)

Berechnungsgraph, DAG. PyTorch baut beim Vorwärtsrechnen einen gerichteten azyklischen Graphen der Operationen; `backward` läuft ihn rückwärts ab und schreibt Gradienten in `.grad`. Die Eingabe ist nur ein weiteres Blatt, deshalb geht Gradientenabstieg auch im Eingaberaum. [Autograd und Berechnungsgraph](#)

requires_grad, Einfrieren. Nur Tensoren mit `requires_grad` bekommen Gradienten und kosten Speicher für den Graphen. Ein vortrainiertes Modell einfrieren heißt, seine Parameter davon auszunehmen und nur den Kopf zu trainieren. [Autograd und Berechnungsgraph](#)

Geteilte Gewichte. Kommt ein Gewicht an mehreren Stellen vor (Faltung, rekurrente Netze), ist sein Gradient die Summe der Beiträge aller Stellen. [Autograd und Berechnungsgraph](#)

6.1.6 Aktivierungsfunktionen und Universalität

ReLU und ihre Ableitung. $\max(0, x)$ mit schwacher Ableitung $[x > 0]$. Billig, kein Sättigen im Positiven. [Aktivierungsfunktionen](#)

Vanishing Gradient. Sigmoid hat Ableitung höchstens $\frac{1}{4}$; über viele Schichten multipliziert verschwindet der Gradient. ReLU vermeidet das im Positiven. [Aktivierungsfunktionen](#)

Sterbende Neuronen, Swish, GELU. Ein ReLU-Neuron mit negativer Aktivierung für alle Daten bekommt nie wieder einen Gradienten. Leaky ReLU, Swish (SiLU) und GELU lassen im Negativen ein kleines Signal durch. [Aktivierungsfunktionen](#)

Universelle Approximation. Ein Netz mit einer versteckten Schicht kann jede stetige Funktion auf einem beschränkten Bereich beliebig gut annähern, aus Treppen- oder Knickfunktionen. Sagt nichts darüber, ob der Gradientenabstieg die Gewichte findet, und nichts über Extrapolation. [Universelle Approximation](#)

Dropout, Batch Normalization. Dropout setzt im Training zufällig Aktivierungen auf null, gegen Overfitting. Batch Normalization normiert die Ausgabe einer Schicht über den Batch auf Mittelwert null und Varianz eins, mit lernbarer Rückskalierung. [Autoencoder](#)

6.1.7 Autoencoder und latente Räume

Autoencoder. Encoder von D auf $L \ll D$ Dimensionen, Decoder zurück, Loss ist die Rekonstruktion (MSE). Approximiert die Identität durch einen Flaschenhals, braucht keine Label, komprimiert. Der Decoder erhöht die Dimension. Linear ist er PCA. [Autoencoder](#)

Standard-AE ist nicht generativ, VAE schon. Der klassische Autoencoder lässt Löcher im latenten Raum; der VAE zwingt die Codes zur Standardnormalverteilung, danach kann man sampeln und dekodieren. [Variational Autoencoder](#)

Denoising AE. Rauschen auf die *Eingabe*, Loss gegen das *saubere* Bild. [Autoencoder](#)

Sparse AE. Strafterm, der die mittlere Aktivierung jeder latenten Dimension klein hält; muss zum Loss addiert werden. [Autoencoder](#)

Weight Decay, L2-Regularisierung. $\lambda \|W\|^2$ zum Loss, zieht Gewichte Richtung null; in PyTorch ein Parameter des Optimierers. [Autoencoder](#)

Early Stopping, Data Augmentation. Aufhören, wenn der Test-Loss wieder steigt; Trainingsdaten durch labelerhaltende Transformationen vervielfachen. [Autoencoder](#)

Reparametrisierungstrick. Statt aus $\mathcal{N}(\mu_x, \sigma_x^2)$ zu sampeln, sample $\varepsilon \sim \mathcal{N}(0, 1)$ und setze $y = \mu_x + \sigma_x \varepsilon$; dann ist ε eine Konstante und der Gradient fließt durch μ_x und σ_x . [Variational Autoencoder](#)

logvar. Der Encoder gibt $\ln \sigma_x^2$ aus statt σ_x , damit kein Ausgang auf positive Werte eingeschränkt werden muss. [Variational Autoencoder](#)

Latent Space, Manifold-Hypothese. Die Daten liegen nahe einer niedrigdimensionalen gekrümmten Fläche; der latente Raum parametrisiert sie. Gerade Linien zwischen Codes können die Fläche verlassen: Interpolation gut, Extrapolation schlecht. [Latente Räume geometrisch](#)

6.1.8 Text, Graphen, Sequenzen

word2vec: CBOW und Skip-gram. CBOW sagt aus dem Kontext das Wort voraus, Skip-gram aus dem Wort den Kontext. Die eine Gewichtsmatrix ist das Embedding. [word2vec](#)

Embedding-Layer. Eine Lookup-Tabelle, die einer ID (Wort, Item, Nutzer) einen gelernten Vektor zuordnet, `nn.Embedding`. [word2vec](#)

Temperatur- und Top-k-Sampling. Beim Generieren die Logits durch T teilen (klein: gieriger, groß: zufälliger) oder nur unter den k wahrscheinlichsten Token sampeln. [Latente Räume geometrisch](#)

node2vec. Random Walks auf einem Graphen erzeugen „Sätze“, darauf word2vec. So bekommen Knoten Vektoren, etwa Items eines Empfehlungssystems. [Representation Learning und Graphen](#)

RNN, GRU, LSTM. Ein rekurrentes Netz verarbeitet eine Sequenz Schritt für Schritt und trägt einen Zustand mit, deshalb geeignet für Zeitreihen. LSTM und GRU steuern mit Gates, was in den Zustand geschrieben und gelesen wird, gegen verschwindende Gradienten über lange Sequenzen. [RNNs](#)

Transformer, Attention. Kein rekurrentes Netz: Jedes Token vergleicht seine Query mit den Keys aller anderen und mischt deren Values, alles parallel. Positionsinformation kommt über ein Positionsempbedding, gelernt oder fest. [Transformer](#)

BERT und GPT. BERT ist ein Encoder, trainiert mit maskierten Wörtern (Masked Language Model), liefert kontextabhängige Embeddings, die man als Features klassifizieren kann. GPT ist ein Decoder, trainiert auf das nächste Token. [Transformer](#)

6.1.9 Faltung und CNNs

Faltung. An jeder Stelle ein Skalarprodukt des Filters mit einem gespiegelten Ausschnitt des Signals. Ein Faltungslayer ist ein spezieller linearer Layer mit vielen Nullen und geteilten Gewichten. [Faltung und CNNs](#)

Parameter Sharing, Translationsäquivalanz. Derselbe Filter an jeder Position: wenige Parameter, und ein verschobenes Bild gibt eine verschobene Feature Map. [Faltung und CNNs](#)

Ausgabegröße. $\lfloor (n + 2p - k) / s \rfloor + 1$ für Eingabe n , Padding p , Kernel k , Stride s . Stride und Pooling verkleinern die Feature Map. Vor dem linearen Kopf **Flatten**. [Faltung und CNNs](#)

Pooling. Max- oder Average-Pooling verkleinert ohne Parameter und macht das Netz robust gegen kleine Verschiebungen. [Faltung und CNNs](#)

1×1-Faltung. Ändert die Kanalzahl, nicht die räumliche Größe: eine lineare Abbildung pro Pixel. [Faltung und CNNs](#)

Gauß-Kernel. Glättet (Tiefpass). Frühe CNN-Schichten lernen von selbst Kanten- und Balkendetektoren. [Faltung und CNNs](#)

6.1.10 Vortrainierte Modelle, LLMs, Destillation

Transfer Learning, Finetuning. Ein auf großen Daten (ImageNet, COCO) vortrainiertes Modell als Startpunkt: Features einfrieren und nur den Kopf trainieren oder alle Gewichte mit kleiner Lernrate nachtrainieren. [Vortrainierte Modelle, LLMs benutzen](#)

Object Detection, Segmentierung. Detection liefert Klassen mit Bounding Boxes, der Loss kombiniert Klassifikation und Box-Regression; Segmentierung liefert pro Pixel eine Klasse. [Vortrainierte Modelle](#)

Lokale LLMs, Quantisierung. Mit Ollama oder llama.cpp laufen Modelle mit wenigen Milliarden Parametern in 4-Bit-Quantisierung auf der CPU. Ein LLM liefert eine Verteilung über dem Vokabular, die man auslesen, als Vektor weiterverarbeiten oder per Constrained Generation einschränken kann. [LLMs benutzen](#)

Destillation. Ein kleines Schülernetz lernt per Regression die (mit Temperatur aufgeweichten) Ausgaben eines großen Lehrers, nicht nur die harten Labels. Speculative Decoding nutzt ein kleines Modell zum Vorschlagen und das große zum Prüfen. [Destillation](#)

Multimodal, Encoder-Decoder. Ein Modell mit Encoder für die eine Modalität und Decoder für die andere (Bild zu Text, Text zu Bild); die Verbindung läuft über einen gemeinsamen latenten Raum. [Vortrainierte Modelle](#)

6.1.11 GANs

Generator und Diskriminator. Der Generator bekommt Rauschen, nie Daten, und sieht die Daten nur über den Gradienten des Diskriminators. Der Diskriminator schätzt die Wahrscheinlichkeit, dass ein Sample echt ist. Beide werden abwechselnd mit Backpropagation trainiert. [GANs](#)

Mode Collapse, Wasserstein. Mode Collapse: der Generator produziert nur noch wenige Varianten. Wasserstein-GANs ersetzen die Kreuzentropie durch eine Transportdistanz mit einem Kritiker, der reelle Zahlen ausgibt; das gibt auch bei nicht überlappenden Verteilungen einen Gradienten. [GANs](#)

6.1.12 Anwendungen

Empfehlungssystem im Latent Space. Items als Vektoren, sodass gemeinsam gekaufte Items nahe liegen (word2vec auf Kauflisten, node2vec, Two-Tower); empfohlen wird, was nahe an den bisherigen Items liegt. [Anwendungsbeispiele](#)

Anwendungskonzept. Was haben wir, was soll rauskommen, welches Modell und woher, welcher Loss, womit messen wir Güte, was kann schiefgehen. [Anwendungsbeispiele](#)

Adversarial Example. Eine für Menschen unsichtbare Änderung der Eingabe, die die Klassifikation kippt; gefunden per Gradientenabstieg im Eingaberaum. [Adversarial Attacks und Polysemie](#)