Programmieren mit Neuronalen Netzen

Michael Gabler

11. Juli 2019

Inhaltsverzeichnis

1 Grundlagen

Neuronales Netzwerk ist Funktion, die auf Eingabedaten angewendet wird.
Optimierung durch Minimierung der Loss-Funktion
Loss-Funktion Maß, wie gut das Netzwerk Vorhersagen trifft. Berechnet sich aus Vorhersage und tatsächlichen Werten (Ground Truth).

- Euklidischer Loss, Mean-Squared-Error: $l_2 = \frac{1}{2N}\sum_i (f_\theta(x_i) - t_i)^2$
- Negative-Log-Likelihood, Cross-Entropy: $NLL = -\frac{1}{|D|}\sum_i \log[f_\theta(x_i)|_{t_i}]$

Konfusionsmatrix welche Klassen werden wie oft mit welcher Klasse verwechselt?

1.1 Training

Daten werden aufgeteilt in Train/Validierung/Test (z.B. 60/20/20)
Epoche Verarbeitung aller Trainingsdaten
Iteration Verarbeitung eines Batches
Batch Mehrere Trainingsbeispiele werden gerechnet bevor Gewichte einmal geupdated werden (z.B. 10 Beispiele pro Batch)
Learning Rate Faktor η, wie stark das Netzwerk durch die Deltas verändert werden soll (d.h. wie schnell es lernt bzw. seine Meinung ändert). Wird beim Update der Gewichte verwendet. **Evaluation auf Validierungsdaten** zur Anpassung der Hyperparameter (Learning-Rate, Netzstruktur, ...)
Evaluation auf Testdaten einmalig, um Genauigkeit des trainierten Netzes zu ermitteln
Forward-Pass Berechnen des Outputs des Netzwerkes für bestimmte Eingabedaten (z.B. ein Batch)
Backward-Pass Bilden der partiellen Ableitung für jeden Input in jedem Layer und Speichern der Werte als Deltas

Backpropagation am Beispiel

$$L = \frac{1}{2}\sum_{i=1}^{2}(y_i - t_i)^2$$

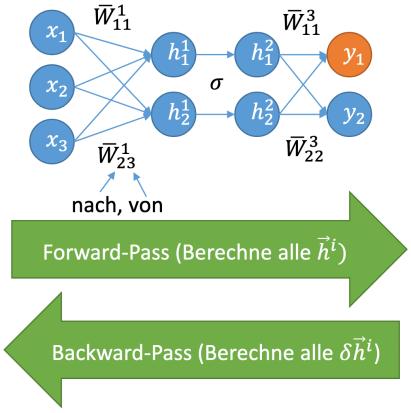
- Gesucht: Fehler an Knoten y_1:

$$\delta y_1 \overset{\text{def}}{=} \frac{\partial L}{\partial y_1} = y_1 - t_1$$

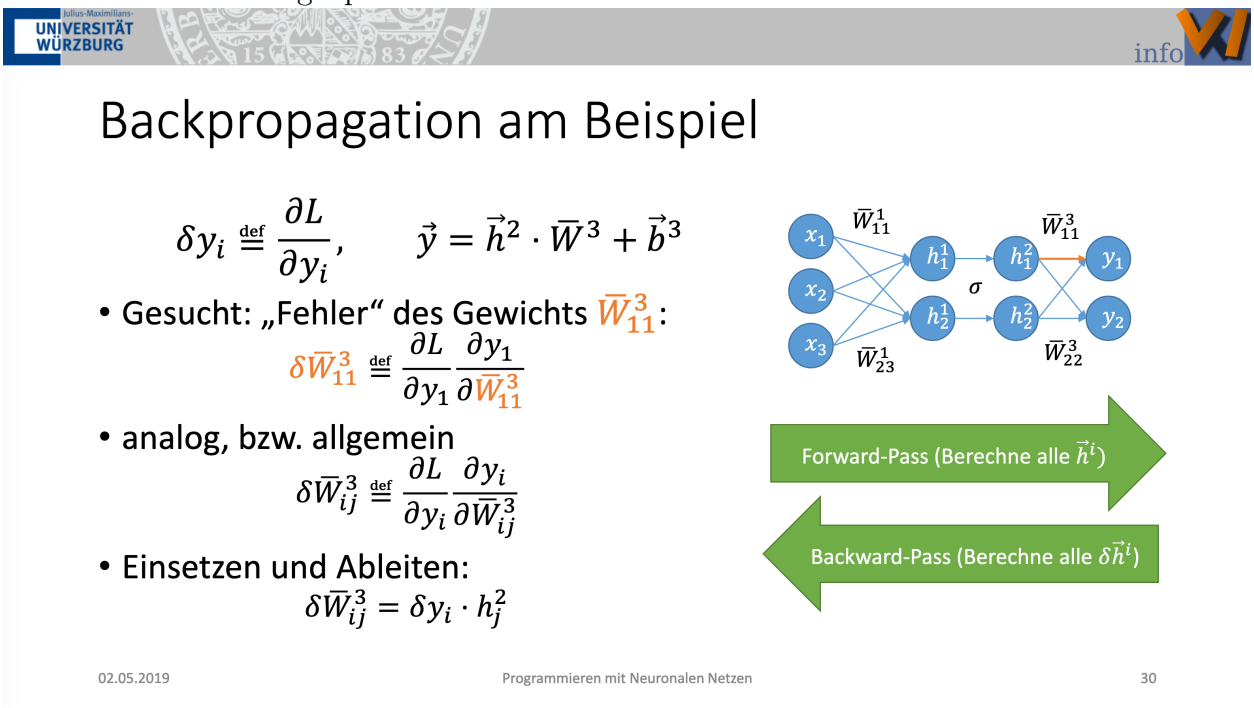
- analog, bzw. allgemein

$$\delta y_i \overset{\text{def}}{=} \frac{\partial L}{\partial y_i} = y_i - t_i$$

Entspricht dem „Fehler“ des Netzwerks



Berechnung der Gewicht-Deltas Bilden der partiellen Ableitung für jedes Gewicht jedes Layers und Speichern der Werte als Deltas. Zur Berechnung sind die Deltas der Outputs (siehe Backward-Pass) erforderlich. Für Batches werden die Deltas der Gewichte aufsummiert und nach dem Batch geupdated.

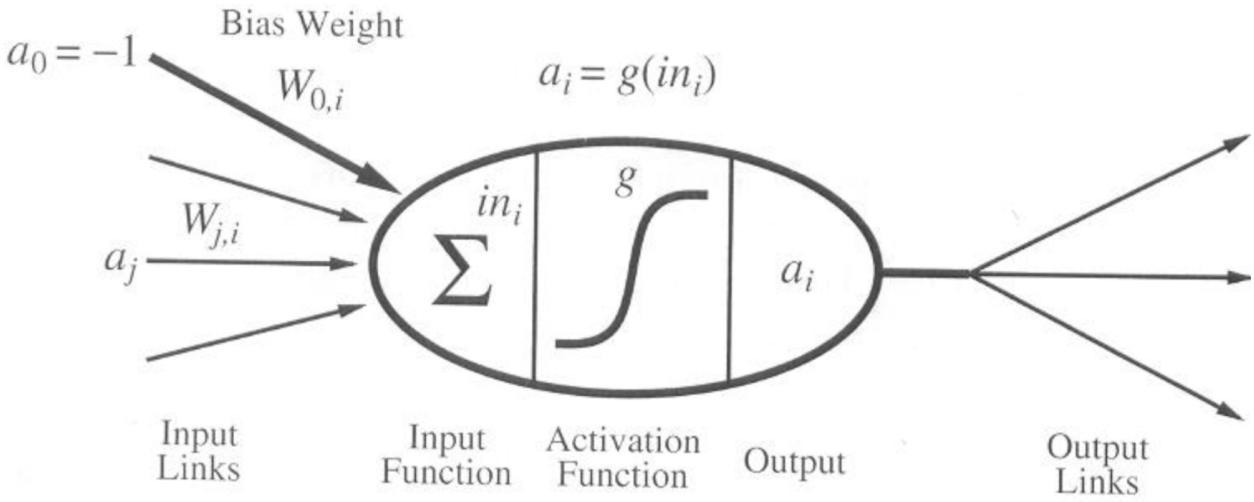


Update der Gewichte Die Gewichte werden nun geupdated, in dem die Deltas der Gewichte mit den ursprünglichen Gewichten verrechnet werden. Dafür gibt es verschiedene Optimierungsverfahren:

- Gradient Descent: für Gewicht $w'_{ij} = w_{ij} - \eta \cdot \delta w_{ij}$
- Adam-Optimizer: robuster gegeüber schlecht gewählter Learning-Rate

1.2 Netzarchitektur

Perceptron (= künstliches Neuron) stellt lineare Trennung (binäre Klassifikation) dar. Kann Funktionen AND, OR und NOT lernen, nicht aber XOR.



mit Inputs a_j gewichtet mit W_{ji} (ergeben zusammen die Gewichtsmatrix W), addiert mit Bias b, Outputs a_i und der Aktivierungsfunktion g mit der der Output berechnet wird.

$$Ausgabe_{Schicht(x)} = g(Eingabe_{Schicht(x)} \cdot W + b) = Eingabe_{Schicht(x+1)}$$

Aktivierungsfunktion muss bei Multi-Layer-Perceptronen (MLP) nicht-linear sein, da sonst nicht mehr Information gespeichert werden kann.

$$(x \cdot W + b) \cdot V + a = x \cdot W \cdot V + b \cdot V + a = x \cdot W' + b'$$

2 Layer

Mögliche Schichten aus denen ein neuronales Netz bestehen kann.

2.1 Fully Connected

Beschreibung
Funktionsweise Bild
Forward-Pass
Backward-Pass
Calculate Delta Weights

2.2 Aktivierungsfunktion

Aktivierungsfunktionen mit Ableitungen

2.3 Softmax

2.4 Dropout

Für Regularisierung verwendet

2.5 Convolutional

2.6 Pooling

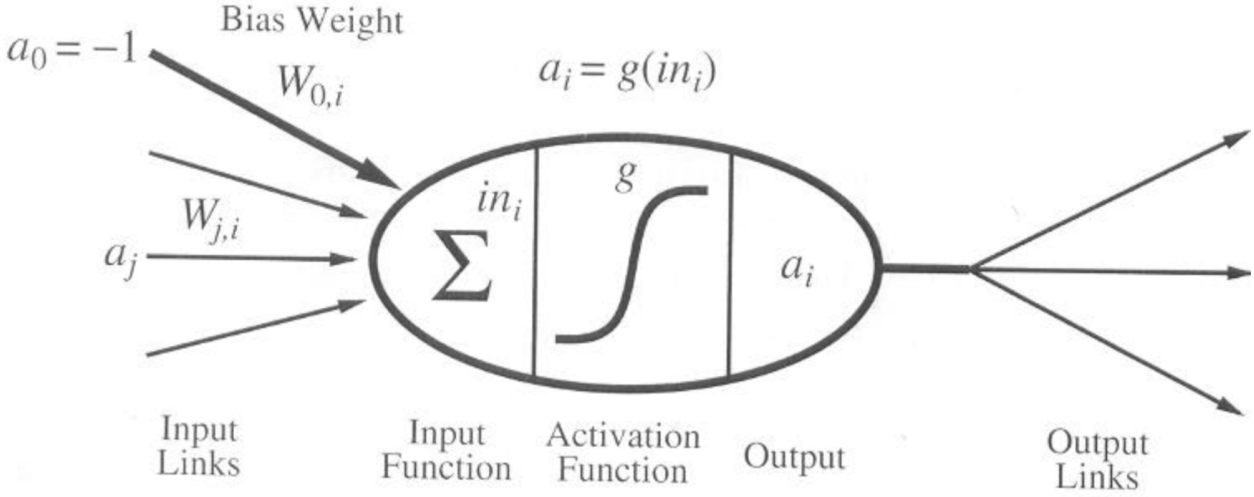
2.7 Vanilla RNN

2.8 LSTM

2.9 Gru

3 aus alter Vorlesung

Perceptron (= künstliches Neuron) stellt lineare Trennung (binäre Klassifikation) dar. Kann Funktionen AND, OR und NOT lernen, nicht aber XOR.



mit Inputs a_j gewichtet mit W_{ji} (ergeben zusammen die Gewichtsmatrix W), Outputs a_i und der Aktivierungsfunktion g mit der der Output berechnet wird.
Bias ist der Input eines Perceptrons, der immer konstant bleibt (meist 1) damit das Perceptron auch bei schwachem Input nicht abgeschaltet wird.
Aktivierungsfunktionen typisch sind:

- Stufenfunktion: $g(x) = x < 0 \to 0, x > 0 \to 1$ (nicht differenzierbar)
- Sigmoidfunktion: $g(x) = \frac{1}{1+e^{-x}}$ (siehe Einleitung Lernen)
- ReLU (Rectified Linear Unit): $g(x) = x < 0 \to 0, x > 0 \to x$ (nicht differenzierbar für $x = 0$)
- Tangenz hyperbolicus: $g(x) = tanh(x)$
- Softmax: Abbildung eines Vektors in Werte zwischen 0 und 1 mit Summe 1 (z.B. für Ausgabelayer)

Loss-Funktion Liefert einen Wert, der repräsentiert, wie gut ein neuronales Netz bereits trainiert ist. Wird anhand des Abgleichs von Vorhersage und tatsächlichem Wert (Ground

Truth) berechnet.
Berechnung der Ausgabe

$$Ausgabe_{Schicht(x)} = g(W \cdot Eingabe_{Schicht(x)}) = Eingabe_{Schicht(x+1)}$$

Rückpropagierung Fehler (z.B. quadratische Abweichung von erwartetem Wert → Loss-Funktion) wird entsprechend der Gewichte auf Vorgängerknoten verteilt
Verteilung des Fehlers

$$Fehler_{Schicht(x-1)} = W^T \cdot Fehler_{Schicht(x)}$$

Änderung der Gewichte über Gradient Descent $w_{jk} \leftarrow w_{jk} - \alpha \frac{dE}{dw_{jk}}$ mit E als Loss-Funktion
Datenvorverarbeitung Inputs zwischen 0 und 1, da hohe Werte niedrige Gradienten erzeugen, Bilder können rotiert, gespiegelt, zugeschnitten, etc. werden um mehr Trainingsdaten zu generieren.
Netzvorbereitung zufällige Vorbelegung der Gewichte (nicht symmetrisch) ⇒ Pre-Training: wenn möglich Gewichte eines bereits trainierten Netzes für ähnliches Problem verwenden
Overfitting kann durch kleine Netzgröße (nicht so ausdrucksstark), Dropout (ignorieren einiger Knoten für einen Lerndurchgang)
Typen von Schichten

- Fully Connected (FC): besteht aus mehreren Perceptrons (s.o.)
- Convolutional Neural Networks (CNN): Für Bilderkennung, Gewichtungsmatrix über Input verschieben und gewichtete, aufsummierte Werte in Outputmatrix schreiben
- Polling Layer: Verkleinerung von Inputmatrizen durch z.B. Max-Funktion einer Submatrix

Multi-Klassen Klassifizierung über K Diskrimianzfunktionen für K Klassen. Dabei gibt immer eine Funktion für einen bestimmten Bereich/Klasse den höchsten Wert zurück.
Vanishing Problem Tiefe Netze lernen sehr langsam, da das Änderungsgewicht pro Schicht mit Faktor ¡ 1 multipliziert wird (Ableitung Aktivierungsfunktion). Kann umgangen werden durch ReLU (Ableitung konstant 1 für positive Zahlen)
Recurrent Neural Networks für die Verarbeitung von sequenziellen Daten (variable Ein-/Ausgabe) z.B. für OCR oder Sprachverarbeitung
Adversial Examples Manipulierte Inputdaten, die für den Menschen noch gut erkennbar sind (kein merklicher Unterschied zu Original), von einem DNN jedoch falsch klassifiziert werden. Bild wird mit Rauschen addiert, welches auf das Netz trainiert ist. Gegenmaßnahmen:

- Trainiere Netz mit Adversial Examples (richtige Klassifikation)
- Mehrere unterschiedliche Technologien zur Klassifikation einsetzen
- DeepCloak entfernen von unnötigen Features, die ausgenutzt werden könnten