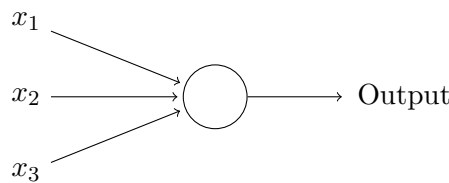
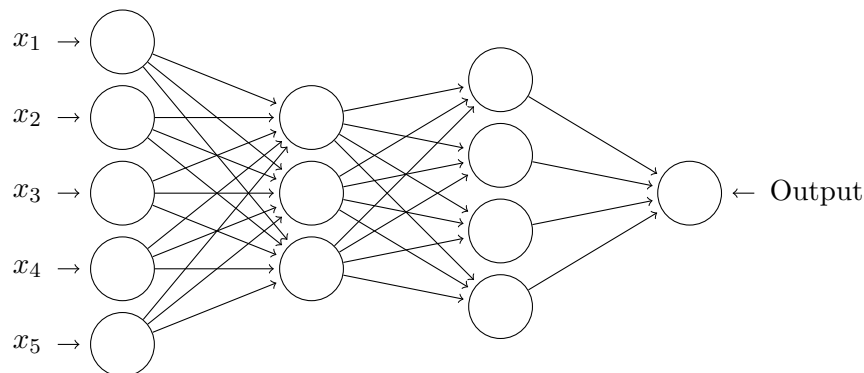


1 Perceptrons

- Modell eines künstlichen Neurons
- Vorgänger der *Sigmoid Neurons*, die in heutigen modernen neuronalen Netzen benutzt werden



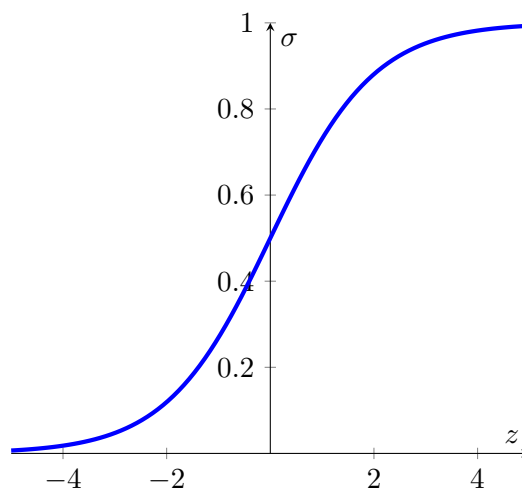
- Eingaben: $x_1, x_2, \dots, x_n \in \{0, 1\}$
- Weights: $w_1, w_2, \dots, w_n \in \mathbb{R}$ für jede Eingabe x_i , der die jeweilige Eingabe gewichtet
- Output =
$$\begin{cases} 0, & \text{wenn } \sum_j w_j x_j \leq \text{Threshold, wobei Threshold} \in \mathbb{R} \\ 1, & \text{sonst} \end{cases}$$
- **vereinfachte Schreibweise:**
 - $\sum_j w_j x_j = w \cdot x$, wobei w und x nun Vektoren beschreiben, dessen Komponenten die Gewichte und Eingaben sind
 - ziehe den Threshold auf die andere Seite der Ungleichung (*Bias*: $b = -\text{Threshold}$)
 - \Rightarrow Bias beschreibt, wie einfach es ist ein Perceptron auf 1 zu bringen
 - Output =
$$\begin{cases} 0, & \text{wenn } w \cdot x + b \leq 0 \\ 1, & \text{sonst} \end{cases}$$
- Mit Hilfe eines *neuronalen Netzes* aus Perceptrons können kompliziertere Entscheidungen getroffen werden:



- neuronales Netz besteht aus drei Schichten: *Input-Layer*, *Hidden-Layer* und *Output-Layer*
- Ziel: bringe das Netz dazu zu lernen, d.h. ihre Weights und Bias-Werte anzupassen, sodass für jede Eingabe das erwartete Ergebnis erzielt wird

2 Sigmoid Neurons

- Anforderung: Eine kleine Änderung in den Weights/Bias-Werten führt nur zu einer kleinen Änderung in der Ausgabe
- \Rightarrow Perceptrons sind dafür nicht geeignet, da sie nur flippen können
- *Sigmoid Neurons*:
 - Eingaben: $x_1, x_2, \dots, x_n \in [0, 1]$
 - Ausgabe: $\sigma(w \cdot x + b) = \sigma(z) = \frac{1}{1+e^{-z}}$
 - Ähnlichkeit: $z \rightarrow \infty \Rightarrow \sigma(z) \approx 1$ und $z \rightarrow -\infty \Rightarrow \sigma(z) \approx 0$



- **Beispiel**: Schrifterkennung
 - Eingabe: $28 \text{ Pixel} \times 28 \text{ Pixel} = 784$ Neuronen mit Intensität $\in [0, 1]$
 - Ausgabe: 10 Neuronen, die die Wahrscheinlichkeiten beschreiben, dass das Bild die entsprechende Zahl zeigt, d.h. Output > 0.5
 - Ziel: approximiere die Funktion $y(x)$, die die Trainingsdaten beschreibt, d.h. $y(x) = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)^T$ für ein Bild x mit einer Null, usw.
 - Methode: minimiere Kostenfunktion $C(w, b) = \frac{1}{2n} \sum_x \|y(x) - a(x)\|^2$, wobei $a(x)$ der Output des neuronalen Netzes bei Input x ist

2.1 Gradient descent

- Ziel: Lösung des Minimierungsproblems $C(v) = C(v_1, v_2, \dots, v_n)$
- Lösung:
 - definiere den Gradienten $\nabla C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \dots, \frac{\partial C}{\partial v_n})^T$ von C aus den partiellen Ableitungen $\frac{\partial C}{\partial v_i}$ von C
 - $\Rightarrow \Delta C \approx \nabla C \cdot \Delta v$
 - wähle $\Delta v = -\eta \nabla C$, dann gilt $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$

warum
noch
mal?

- * η wird das *Lerntempo* (engl. *learning rate*) genannt
- * da $\|\nabla C\|^2 \geq 0$ folgt, dass $\Delta C \leq 0$ für alle $\eta \in \mathbb{R}^+$
- * wähle η so, dass wir nicht zu langsam lernen, aber dennoch eine gute Approximation erhalten

2.1.1 Gradient descent in neuronalen Netzen

- ∇C besteht aus partiellen Ableitungen der Komponenten w_k und b_l
- Problem: ∇C wird berechnet aus dem Mittelwert der Gradienten $\nabla C_x = \frac{1}{2} \|y(x) - a(x)\|^2$ für alle Trainingsdaten $x \Rightarrow$ berechnungsintensiv
- Lösung: *Stochastic gradient descent*
 - Idee: berechne ∇C aus einer kleinen Anzahl m zufällig gewählter Trainingsdaten X_1, X_2, \dots, X_m
 - vorausgesetzt m ist groß genug, dann gilt $\frac{1}{m} \sum_{i=1}^m \nabla C_{X_i} \approx \frac{1}{n} \sum_x \nabla C_x = \nabla C$

3 Backpropagation

- berechnet den Gradienten ∇C_x der Kostenfunktion C_x

3.1 Notation

- w_{jk}^l beschreibt das Gewicht des k ten Neurons im $(l-1)$ ten Layer zu dem j ten Neuron im l ten Layer
- b_j^l beschreibt den Bias des j ten Neurons im l ten Layer
- a_j^l beschreibt die Ausgabe (*Aktivierung*) des j ten Neurons im l ten Layer
- $\Rightarrow a_j^l = \sigma((\sum_k w_{jk}^l a_k^{l-1}) + b_j^l)$
- Intuition: lässt uns die Aktivierung und den Bias als Vektor schreiben, die Gewichte zwischen zwei Layern als Matrix

– Beispiel:

- * Layer 1 besitzt 4 Neuronen, Layer 2 besitzt 3 Neuronen

$$* a^1 = \begin{pmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \\ a_4^1 \end{pmatrix}, b^2 = \begin{pmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{pmatrix}, w^2 = \begin{pmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 & w_{14}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 & w_{24}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 & w_{34}^2 \end{pmatrix}$$

- * $a^2 = \sigma(w^2 \cdot a^1 + b^2)$, wobei σ auf jeder Komponente einzeln angewendet wird

- $z^l \equiv w^l a^{l-1} + b^l$ wird als *gewichtete Eingabe* bezeichnet
- $\Rightarrow C_x = \frac{1}{2} \|y(x) - a^L(x)\|^2$, wobei $a^L(x)$ der Vektor des letzten Layers bei Eingabe x beschreibt

3.1.1 Hadamard Produkt $s \odot t$

- elementweise Multiplikation zweier Vektoren
- Beispiel: $\begin{pmatrix} 1 \\ 2 \end{pmatrix} \odot \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \cdot 3 \\ 2 \cdot 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \end{pmatrix}$

3.2 Die vier fundamentalen Gleichungen der Backpropagation

- $\delta_j^l \equiv \frac{\partial C_x}{\partial z_j^l}$ bezeichnet den *Fehler* im j ten Neuron im l ten Layer
 - Veranschaulichung:
 1. an dem j ten Neuron im l ten Layer kommt eine Eingabe z_j^l an
 2. es wird eine kleine Veränderung Δz_j^l vorgenommen, sodass die Kosten minimiert werden
 - * wenn $\frac{\partial C_x}{\partial z_j^l}$ einen großen (negativen) Wert hat, dann können die Kosten reduziert werden, wenn Δz_j^l entgegengesetzt zur Steigung gewählt wird
 - * wenn $\frac{\partial C_x}{\partial z_j^l}$ nahe an Null, dann ist auch Δz_j^l nahe bei Null
 3. anstatt $\sigma(z_j^l)$ wird $\sigma(z_j^l + \Delta z_j^l)$ weitergeleitet
- Backpropagation berechnet δ_j^l und projiziert den Fehler auf $\frac{\partial C_x}{\partial w_{jk}^l}$ und $\frac{\partial C_x}{\partial b_j^l}$

1. Der Fehler δ^L im Output-Layer:

- die Komponenten von δ^L sind gegeben durch $\delta_j^L = \frac{\partial C_x}{\partial a_j^L} \sigma'(z_j^L)$
- $\frac{\partial C_x}{\partial a_j^L} = (a_j - y_j)$ beschreibt, wie schnell sich die Kosten in der j ten Ausgabe verändern
- \Rightarrow wenn C_x unabhängig von dem Ausgabeneuron j , dann ist δ_j^L klein
- Matrix-Schreibweise: $\delta^L = \nabla_a C_x \odot \sigma'(z^L)$

2. Der Fehler δ^l in Abhängigkeit zu δ^{l+1} :

- $\delta^l = ((w^{l+1})^T \cdot \delta^{l+1}) \odot \sigma'(z^l)$
- Angenommen, wir kennen den Fehler δ^{l+1}
- das Transponieren der Matrix w^{l+1} kann als rückwärts durch das Netz gehen verstanden werden

3. Abhängigkeit des Fehlers δ_j^l zur Kostenveränderung bzgl. des Bias b_j^l :

- $\frac{\partial C_x}{\partial b_j^l} = \delta_j^l$

4. Abhängigkeit des Fehlers δ_j^l zur Kostenveränderung bzgl. des Gewichts w_{jk}^l :

- $\frac{\partial C_x}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$

3.3 Algorithmus

1. fülle das Netz mit beliebigen Gewichten und Bias-Werten
2. setze x als Eingabe des Netzes
3. berechne z^l und $a^l = \sigma(z^l)$ für jeden Layer
4. bestimme den Output-Error δ^L über die erste Gleichung
5. bestimme den Fehler δ^l rückwärts für jeden Layer l mit $l = \{L-1, L-2, \dots, 2\}$
6. berechne die Gradienten $\frac{\partial C_x}{\partial w_{jk}^l}$ und $\frac{\partial C_x}{\partial b_j^l}$
 - für eine zufällige Anzahl m an Trainingsdaten gilt dann (Stochastic gradient descent):
 1. berechne den Fehler $\delta^{x,l}$ in jeder Schicht l und für jede Eingabe x
 2. verändere die Gewichte anhand folgender Regel:

$$\begin{aligned} - w^l &\rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T \\ - b^l &\rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l} \end{aligned}$$

warum?

4 Cross-Entropy

- Netz lernt wohlmöglich ziemlich langsam (braucht viele Iterationen, um die Trainingsdaten gut zu approximieren)
- Netz lernt langsam, wenn $\frac{\partial C}{\partial w}$ und $\frac{\partial C}{\partial b}$ klein
- $\Rightarrow \sigma'(z)$ ist klein für große/kleine z -Werte
- kann verbessert werden, in dem eine andere Kostenfunktion benutzt wird \Rightarrow **Cross-Entropy**:

$$C = -\frac{1}{n} \sum_x y \ln a + (1-y) \ln(1-a)$$

4.1 Eigenschaften

- $C > 0$:
 - $y, a, (1-y), (1-a) \in [0, 1]$