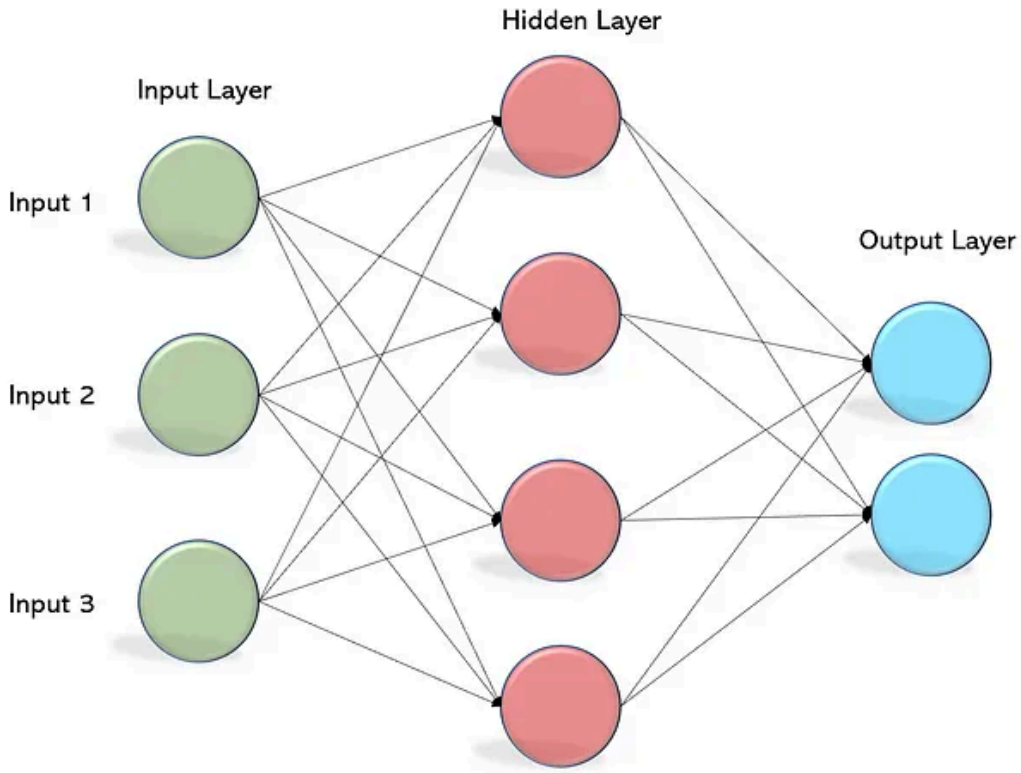
Multi-Layer Perceptron (MLP)

https://www.datacamp.com/de/tutorial/multilayer-perceptrons-in-machine-learning

Ein **Multi-Layer Perceptron (MLP)** ist ein neuronales Netzwerk mit mehreren Schichten:

- **Eingabeschicht** (Input Layer)
- **Eine oder mehrere versteckte Schichten** (Hidden Layers)
- **Ausgabeschicht** (Output Layer)

MLPs sind **Feedforward-Netzwerke**, was bedeutet, dass die Daten nur in eine Richtung fließen: von den Eingaben zu den Ausgaben. Sie verwenden die **Backpropagation**, um die Gewichte zu aktualisieren.



Forward Propagation

Forward Propagation ist der Prozess, bei dem Eingabedaten durch ein **neuronales Netzwerk** geleitet werden, um eine Ausgabe zu berechnen.
Es ist der erste Schritt im Trainings- und Vorhersageprozess eines **Multi-Layer Perceptron (MLP)**. In diesem Schritt werden:

1. Die **Eingaben** in das Netzwerk eingespeist.
2. Die **Neuronen** in jeder Schicht berechnen eine gewichtete Summe der Eingaben plus einen Bias und wenden darauf eine **Aktivierungsfunktion** an.
3. Die Ausgabe der letzten Schicht liefert die finale Vorhersage des Netzwerks.

Prinzip von Forward Propagation

Ein **Multi-Layer Perceptron (MLP)** verarbeitet Informationen schichtweise und besteht aus drei Hauptkomponenten:

1. **Eingabeschicht**
   - Die **Eingabeneuronen** erhalten die Eingangsdaten und leiten sie unverändert an die nächste Schicht weiter.
   - Diese Neuronen führen **keine gewichtete Summe oder Aktivierungsfunktion** aus.
   - Formal repräsentieren die Eingaben **Rohdaten**, die als Vektoren notiert werden:

$$\mathbf{x} = (x_1, x_2, \ldots, x_n)$$

2. **Versteckte Schicht(en)**
   - Jedes Neuron in dieser Schicht berechnet eine gewichtete Summe der Eingaben:

$$z_j = \sum_{i=1}^{n} w_{ji} x_i + b_j$$

   - Anschließend wird eine Aktivierungsfunktion $f(z)$ angewendet:

$$a_j = f(z_j)$$

   - Häufig verwendete Aktivierungsfunktionen:
     - **Sigmoid:** $f(z) = \frac{1}{1+e^{-z}}$
     - **ReLU:** $f(z) = \max(0, z)$
     - **Tanh:** $f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

3. **Ausgabeschicht**

- Diese Schicht nimmt die Ausgaben der letzten versteckten Schicht und verarbeitet sie auf dieselbe Weise:

$$z_k = \sum_{j=1}^{m} w_{kj} a_j + b_k$$

- Eine Aktivierungsfunktion wird angewendet, um die endgültige Vorhersage (y) zu berechnen:

$$y = f(z_k)$$

Beispiel: Einfaches Netzwerk mit einer versteckten Schicht

Wir betrachten ein **MLP mit einer versteckten Schicht** und folgenden Parametern:

- **Eingaben:** x_1, x_2
- **Versteckte Schicht mit 2 Neuronen**
- **1 Ausgabeneuron**
- **Sigmoid-Aktivierung** in allen Schichten

Gegebene Werte

- **Eingaben:**
  $x_1 = 0.5, \quad x_2 = 0.3$
- **Gewichte:**
  - **Versteckte Schicht:**
    $w_{11} = 0.4, \quad w_{12} = 0.2$
    $w_{21} = 0.1, \quad w_{22} = 0.5$
  - **Ausgabeschicht:**
    $w_{31} = 0.3, \quad w_{32} = 0.7$
- **Bias-Werte:**
  $b_1 = 0.1, \quad b_2 = -0.3, \quad b_3 = 0.2$

Berechnung der versteckten Schicht

Jedes Neuron in der versteckten Schicht berechnet zunächst den Nettoeingang:

Neuron 1 in der versteckten Schicht

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1$$

$$z_1 = (0.4 \cdot 0.5) + (0.2 \cdot 0.3) + 0.1$$

$$z_1 = 0.2 + 0.06 + 0.1 = 0.36$$

Die Aktivierung wird mit der **Sigmoid-Funktion** berechnet:

$$a_1 = \frac{1}{1 + e^{-0.36}} \approx 0.59$$

Neuron 2 in der versteckten Schicht

$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2$$

$$z_2 = (0.1 \cdot 0.5) + (0.5 \cdot 0.3) + (-0.3)$$

$$z_2 = 0.05 + 0.15 - 0.3 = -0.1$$

Aktivierung:

$$a_2 = \frac{1}{1 + e^{-(-0.1)}} \approx 0.48$$

Berechnung der Ausgabeschicht

Die beiden Aktivierungen (a_1) und (a_2) sind die Eingaben für das **Ausgabeneuron**.

$$z_3 = w_{31}a_1 + w_{32}a_2 + b_3$$

$$z_3 = (0.3 \cdot 0.59) + (0.7 \cdot 0.48) + 0.2$$

$$z_3 = 0.177 + 0.336 + 0.2 = 0.713$$

Finale Ausgabe mit Sigmoid-Aktivierung:

$$y = \frac{1}{1 + e^{-0.713}} \approx 0.67$$

Fehlerberechnung (Loss Calculation)

Nachdem die **Forward Propagation** abgeschlossen ist und das Netzwerk eine Ausgabe generiert hat, folgt der nächste Schritt: die **Fehlerberechnung**.

Allgemeine Beschreibung der Fehlerberechnung

Das Ziel eines neuronalen Netzwerks ist es, die vorhergesagte Ausgabe $\hat{y}$ so nah wie möglich an die tatsächliche Zielausgabe y zu bringen.
Die **Fehlerberechnung** erfolgt mithilfe einer **Kostenfunktion (Loss Function)**, die misst, wie weit die Vorhersage vom wahren Wert abweicht.

Kostenfunktion (Loss Function)

Eine **Kostenfunktion** ordnet jeder Netzwerkausgabe einen Fehlerwert zu.
Je größer der Fehlerwert, desto schlechter ist die Vorhersage.

Je nach Problemstellung gibt es verschiedene Kostenfunktionen:

1. **Mittlere quadratische Abweichung (Mean Squared Error, MSE)**
    - Wird für **Regressionsprobleme** verwendet.
    - Berechnet den Durchschnitt der quadrierten Fehler:

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

2. **Kreuzentropie (Cross-Entropy Loss)**
    - Wird für **Klassifikationsprobleme** verwendet.
      Falsche, aber **unsichere Vorhersagen** werden **weniger** bestraft als **sichere falsche Vorhersagen**.

$$J = -\sum y \log(\hat{y})$$

- Beispiel für **binäre Klassifikation** (zwei Klassen):

$$J = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Beispiel: Fehlerberechnung für unser MLP

Wir verwenden das gleiche **MLP-Netzwerk mit einer versteckten Schicht**, das wir für Forward Propagation genutzt haben.

Aus der Forward Propagation haben wir die folgende Ausgabe berechnet:

$$\hat{y} \approx 0.67$$

Nehmen wir an, der gewünschte Zielwert ist:

$$y = 1$$

Nun berechnen wir den Fehler mit der **Mean Squared Error (MSE)**-Funktion:

Berechnung der MSE-Kostenfunktion

$$J = \frac{1}{2}(y - \hat{y})^2 = J = \frac{1}{2}(1 - 0.67)^2 \approx 0.054$$

Berechnung mit Cross-Entropy Loss

Falls es sich um ein **binäres Klassifikationsproblem** handelt, verwenden wir die **Binary Cross-Entropy Loss**:

$$J = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] = J = -[1 \log(0.67) + 0 \log(1 - 0.67)] \approx 0.4$$

Backpropagation

Nachdem wir in der **Forward Propagation** die Ausgabe des Netzwerks berechnet und in der **Fehlerberechnung (Loss Calculation)** den Fehler bestimmt haben, folgt nun der dritte Schritt: **Backpropagation**.

Was ist Backpropagation?

Backpropagation ist der Prozess, mit dem ein neuronales Netzwerk aus seinen Fehlern lernt. Nachdem das Netzwerk eine Vorhersage gemacht hat, wird der Unterschied zwischen der Vorhersage und dem tatsächlichen Wert berechnet – das ist der Fehler. Dieser Fehler wird dann rückwärts durch das Netzwerk propagiert, um herauszufinden, welche Neuronen und welche Verbindungen für den Fehler verantwortlich sind.

Jede Verbindung im Netzwerk hat ein Gewicht, das bestimmt, wie stark ein Neuron das nächste beeinflusst. Mithilfe der Ableitungen (also der Steigungen der Fehlerfunktion) wird berechnet, wie diese Gewichte verändert werden müssen, um den Fehler beim nächsten Durchlauf zu verringern. Dabei werden zuerst die Anpassungen in der **Ausgabeschicht** berechnet und dann Schritt für Schritt die Schichten davor angepasst, bis zur **versteckten Schicht**.

Dieser Prozess wird viele Male wiederholt – oft für Tausende oder Millionen von Beispielen. Jedes Mal wird der Fehler kleiner, bis das Netzwerk schließlich gute Vorhersagen machen kann. Der Algorithmus nutzt dazu **Gradient Descent**, eine Technik, um schrittweise in Richtung der besten Gewichte zu gehen. So lernt das Netzwerk nach und nach, Muster in den Daten zu erkennen und genauere Ergebnisse zu liefern.

Backpropagation (kurz: **Backprop**) ist der Algorithmus, mit dem das neuronale Netzwerk aus seinen Fehlern lernt.
Er besteht aus zwei Hauptaufgaben:

1. **Berechnung des Gradienten** der Kostenfunktion J bezüglich der Gewichte w.
2. **Aktualisierung der Gewichte**, um den Fehler zu minimieren.

Backpropagation basiert auf der **Kettenregel der Ableitung**, um den Einfluss jeder Gewichtung auf den Fehler zu bestimmen.

Mathematische Grundlagen der Backpropagation

Jede Gewichtung im Netzwerk beeinflusst den Fehler unterschiedlich stark. Um zu lernen, müssen wir herausfinden, wie stark sich eine Änderung des Gewichts w auf die Kostenfunktion J auswirkt.

Das bedeutet, wir berechnen die **Ableitung der Kostenfunktion nach den Gewichten**:

$$\frac{\partial J}{\partial w}$$

Da die Gewichtungen nicht direkt mit J verknüpft sind, nutzen wir die **Kettenregel**, um den Gradienten rückwärts durch das Netzwerk zu propagieren:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Dabei sind:

- J die Kostenfunktion,
- $\hat{y}$ die vorhergesagte Ausgabe,
- z die gewichtete Summe vor der Aktivierungsfunktion.

Der letzte Term:

$$\frac{\partial z}{\partial w} = a$$

ist immer die **Aktivierung des vorherigen Neurons**, und beim Updaten der Gewichte wird dies direkt in die Formel eingebaut (siehe unten)

Gradienten für die Ausgabeschicht

Das letzte Neuron gibt die Vorhersage $\hat{y}$ aus. Die Ableitung der Kostenfunktion nach der Ausgabe ist:

$$\frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

Falls **Sigmoid** als Aktivierungsfunktion genutzt wird:

$$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y})$$

Damit ergibt sich der **Fehler der Ausgabeschicht** als:

$$\delta_{\mathrm{k}} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$$

Gradienten für die versteckte Schicht

Mit der Kettenregel erhalten wir den Gradienten für die versteckte Schicht:

$$\frac{\partial J}{\partial w_{ji}} = \sum_k \left(\frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_k} \cdot \frac{\partial z_k}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ji}} \right)$$

Setzen wir die bekannten Terme ein:

$$\frac{\partial J}{\partial w_{ji}} = \sum_k ((\hat{y} - y) \cdot \hat{y}(1 - \hat{y}) \cdot w_{kj} \cdot a_j(1 - a_j) \cdot a_i)$$

Da der **Fehlerterm der Ausgabeschicht** bereits als δ_k definiert wurde:

$$\delta_k = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$$

können wir den Fehler der versteckten Schicht so schreiben:

$$\delta_j = \sum_k \delta_k w_{kj} \cdot a_j(1 - a_j)$$

Aktualisierung der Gewichte

Nachdem die Gradienten berechnet wurden, werden die Gewichte mit **Gradient Descent** aktualisiert:

$$w = w - \eta \cdot \delta \cdot a$$

- η ist die **Lernrate**, die bestimmt, wie stark die Gewichte angepasst werden.
- δ ist der Fehlerterm.
- a ist die Aktivierung des vorherigen Neurons.

Beispiel: Backpropagation für unser MLP

Wir nehmen unser **MLP mit einer versteckten Schicht** aus der Forward Propagation und berechnen die Gewichtsanpassungen.

Gegebene Werte

- **Zielwert:** $y = 1$
- **Vorhersage:** $\hat{y} = 0.67$
- **Aktivierungen der versteckten Schicht:**

$$a_1 = 0.59, \quad a_2 = 0.48$$

- **Gewichte der Ausgabeschicht:**

$$w_{31} = 0.3, \quad w_{32} = 0.7$$

Fehlerberechnung für die Ausgabeschicht

Der Fehler in der Ausgabeschicht:

$$\delta_{\text{output}} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$$

$$\delta_{\text{output}} = (0.67 - 1) \cdot 0.67(1 - 0.67)$$

$$\delta_{\text{output}} = (-0.33) \cdot (0.67 \cdot 0.33)$$

$$\delta_{\text{output}} \approx -0.073$$

Fehlerberechnung für die versteckte Schicht

Jedes Neuron in der versteckten Schicht erhält einen **Fehleranteil** basierend auf seinen Verbindungen zur Ausgabeschicht.

Für **Neuron 1**:

$$\delta_1 = \delta_{\text{output}} \cdot w_{31} \cdot a_1(1 - a_1)$$

$$\delta_1 = (-0.073) \cdot (0.3) \cdot (0.59 \cdot 0.41)$$

$$\delta_1 \approx -0.0053$$

Für **Neuron 2**:

$$\delta_2 = \delta_{\text{output}} \cdot w_{32} \cdot a_2(1 - a_2)$$

$$\delta_2 = (-0.073) \cdot (0.7) \cdot (0.48 \cdot 0.52)$$

$$\delta_2 \approx -0.0128$$

Gewichtsaktualisierung

Die neuen Gewichte werden berechnet mit:

$$w = w - \eta \cdot \delta \cdot a$$

Angenommen, die **Lernrate** $\eta = 0.1$:

Neue Gewichte der Ausgabeschicht

$$w_{31} = 0.3 - (0.1 \cdot (-0.073) \cdot 0.59)$$

$$w_{31} \approx 0.304$$

$$w_{32} = 0.7 - (0.1 \cdot (-0.073) \cdot 0.48)$$

$$w_{32} \approx 0.703$$

Neue Gewichte der versteckten Schicht

Die Eingaben waren $x_1 = 0.5$, $x_2 = 0.3$:

$$w_{11} = 0.4 - (0.1 \cdot (-0.0053) \cdot 0.5)$$

$$w_{11} \approx 0.401$$

$$w_{12} = 0.2 - (0.1 \cdot (-0.0053) \cdot 0.3)$$

$$w_{12} \approx 0.20016$$

$$w_{21} = 0.1 - (0.1 \cdot (-0.0128) \cdot 0.5)$$

$$w_{21} \approx 0.10064$$

$$w_{22} = 0.5 - (0.1 \cdot (-0.0128) \cdot 0.3)$$

$$w_{22} \approx 0.50038$$

Folgerung: Aktivierungsfunktionen müssen differenzierbar sein !

Gradient Descent basiert darauf, dass wir **die Ableitung der Kostenfunktion** berechnen, um zu wissen, in welche Richtung und wie stark wir die Gewichte ändern müssen.

Da die Aktivierungsfunktionen die Neuronenausgaben beeinflussen, müssen wir ihre Ableitungen berechnen können.
Wenn die Aktivierungsfunktion **nicht differenzierbar** wäre, könnten wir den Gradienten nicht bestimmen und wüssten nicht, wie wir die Gewichte anpassen sollen.

Welche Aktivierungsfunktionen sind differenzierbar?

Die meisten Aktivierungsfunktionen, die in neuronalen Netzwerken verwendet werden, sind **stetig und differenzierbar**. Beispiele:

- **Sigmoid:**

$$f(x) = \frac{1}{1 + e^{-x}}$$

  → Ableitung:

$$f'(x) = f(x)(1 - f(x))$$

- **Tanh:**

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  → Ableitung:

$$f'(x) = 1 - f(x)^2$$

- **ReLU (Rectified Linear Unit):**

$$f(x) = \max(0, x)$$

  → Ableitung:

$$f'(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \\ \text{undefiniert}, & x = 0 \end{cases}$$

Problem: ReLU ist **nicht** differenzierbar bei $x = 0$, aber fast überall sonst. In der Praxis wird das einfach ignoriert.

Welche Funktionen sind problematisch?

- **Schritt- oder Stufenfunktionen**, wie z. B. **Heaviside-Funktion**:

$$f(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

  Diese Funktion hat **keine Ableitung**, weil sie an der Sprungstelle **nicht stetig** ist.

→ Gradient Descent funktioniert hier nicht, weil es keine Information gibt, in welche Richtung sich die Gewichte bewegen sollen.