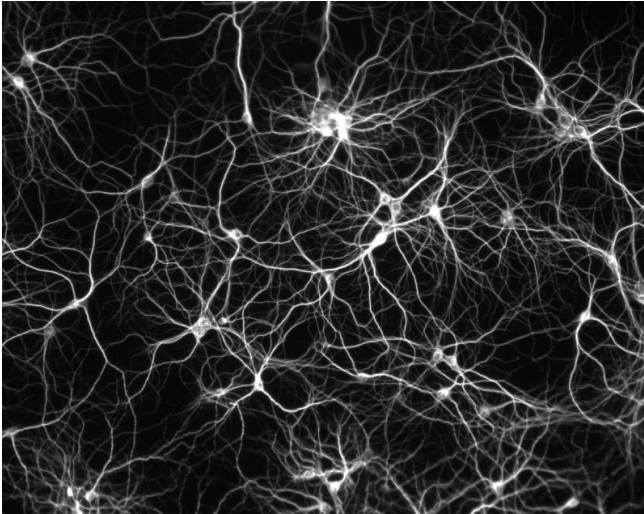


### 3.6.3 Künstliche Neuronale Netze

In Data Mining geht es also viel um Klassifizieren, und wir sollten uns noch an einem speziellen Typ von Klassifikator wenden, der eine spezielle Bedeutung besitzt, der neuronale Netzwerk. Menschen sind, dank der Evolution, sehr gut geeignet zu klassifizieren. Im Gegenteil zu Silizium-Chips brauchen sie viel Zeit zu rechnen, dafür sind in Erkennung von Patterns sowie Clustering (auch eine Art Klassifikation) immer noch viel besser als Silizium. Diese Feststellung hat viele Forschenden veranlasst zu untersuchen, ob man die Funktionsweise des Hirns simulieren könnte. Das wurde mit den neuronalen Netzwerke gemacht. Diese versuchen, in sehr vereinfachte Weise die Funktion des Hirns zu simulieren.



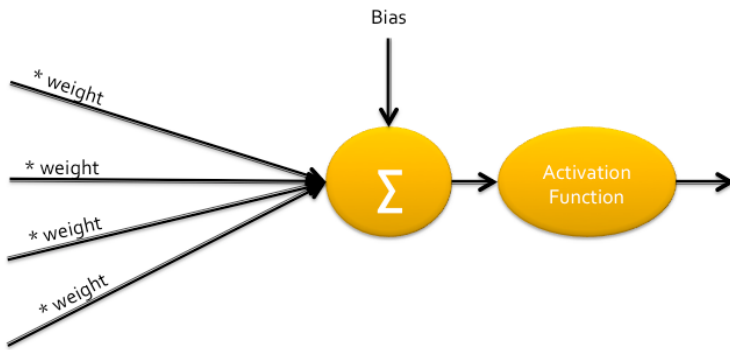
**Abbildung 14: Eine Kultur von Neuronen (aus: cranepsycho.edublogs.org). Man erkennt leicht einzelnen Neuronen (Klumpen) und ihre zusammenverbundene Synapsen („Gewebe“)**

Wie ersichtlich in Abbildung 14 kommt die Funktionalität eines Neurons aus den Verbindungen, die er mit den anderen Neuronen pflegt, eher als durch seinen eigene Funktion als Einzelstück. Die „Business Logic“ ist damit in den Verbindungen eher als in den einzelnen Neuronen „kodierte.“ Damit sind die einzelnen Zellen unwichtig (einer konnte auch ausfallen). Was verstehen wir unter Business Logic in dem Fall? Wir wollen einen Netzwerk aufbauen, der bestimmte Input bearbeitet, und daraus Output erstellt, die uns Information über die Klassen in welchen den Input gehören liefert.

Ein Beispiel kann auf die Daten von Tabelle 5 (Versicherungsdaten) aufgebaut werden. In diesem Datenbank-Schema hatten wir 2 Input-Spalte und eine Output-Spalte. Aus weitem gesehen würde ein neuronales Netzwerk, um diese Daten zu analysieren wie eine Black Box aussehen: Inputs-Verbindungen tragen die sind die Inputwerte in dem Box, und aus dem Box kommen Output-Verbindungen heraus, die Klassen der eingegangenen Werten ermitteln. Wie in den vorherigen Abschnitte wird hier auch zuerst mit einem a priori bekannten Datenmenge das Netzwerk trainiert, bevor es im Einsatz kommen kann, um mit allgemeinen Werte die Klassifizierung bestimmen kann.

#### 3.6.3.1 Aufbau eines künstlichen Neurons

Aus die Beobachtung der biologischen Neuronen probieren wir einen Computermodell zu erstellen, das ein solches biologische Netzwerk imitieren kann. Das künstliche Netzwerk wird selbstverständlich viel einfacher sein als das Originale, da Menschen etwa  $10^{11}$  Neuronen besitzen. Ein einzelnen künstlichen Neuron ist in Abbildung 15 dargestellt.



**Abbildung 15: Schematische Darstellung eines einzelnen künstlichen Neuron.**

Dieses Neuron wurde schon 1943 von McCulloch & Pitts vorgeschlagen. Es ist eine binäre Anordnung: es berechnet zuerst eine gewichtete Summe seiner Inputs, und feuert über einem Schwellwert einen Signal heraus (die Bedeutung von Bias wird später besprochen):

$$n_i(t+1) = \vartheta\left(\sum w_{i,j}n_j(t) - \mu_i\right)$$

In dieser Gleichung ist  $n_i(t)$  der Zustand des Neurons zur Zeit  $t$ . In dieser Anordnung ist die Zeit als eine stetig wachsende Ganzzahl (Integer) zu verstehen. Die Funktion  $\vartheta(x)$  ist die Stufenfunktion, die den Output-Wert bestimmt:

$$\vartheta(x) = 1 \text{ wenn } x \geq 0,$$

$$\vartheta(x) = 0 \text{ sonst.}$$

Die Gewichte  $w_{i,j}$  bezeichnen die Stärke der Signalen die aus den anderen Neuronen kommen. Man spricht von stimulierend (excitatory) oder hemmend (inhibitory) Signale, in Abhängigkeit der Wirkung Ihrer Werte auf der gesamten Summe. Der Parameter  $\mu_i$  bezeichnet der Bias, ein pro Neuron frei einstellbare Wert, der den Schwellwert des Neurons bezeichnet, d.h also, die Summe der Signale aus Synapsen muss dieses Wert übersteigen, bevor der Neuron feuern kann.

Auch wenn sehr einfach, ist dieses künstliche Neuron eine sehr leistungsfähiges Bauteil. Theoretisch kann eine Anordnung von solche Bauteile alles berechnen, es ist ein universelles Rechners, wie McCulloch and Pitts gezeigt haben.

Eine selbstverständliche Verallgemeinerung des Bauteils kann wie folgt angegeben werden

$$n_i(t+1) = g\left(\sum w_{i,j}n_j(t) - \mu_i\right)$$

Wobei  $n_i(t)$  jetzt Status oder Aktivtion des Bauteiles genannt wird.  $g(x)$  ist die Aktivationsfunktion, auch noch Gain Function, oder Transfer Function genannt. Die Aktivationsfunktion ist im allgemeinen nichtlinear, aber sie ist nicht auf Stufenfunktionen begrenzt wie  $\vartheta(x)$ . Im Gegenteil andere Arten von Funktionen werden bevorzugt, z.B. Sigmoid-Funktionen oder die hyperbolische Tangentialfunktion, die in Abbildung 16 angegeben ist. Der Grund der Bevorzugung einer Sigmoidalfunktion liegt darin, dass sie dann ableitbar ist, eine Bedingung für die Konvergenz in vielen Lernalgorithmen wie die Backpropagation (siehe nachher).

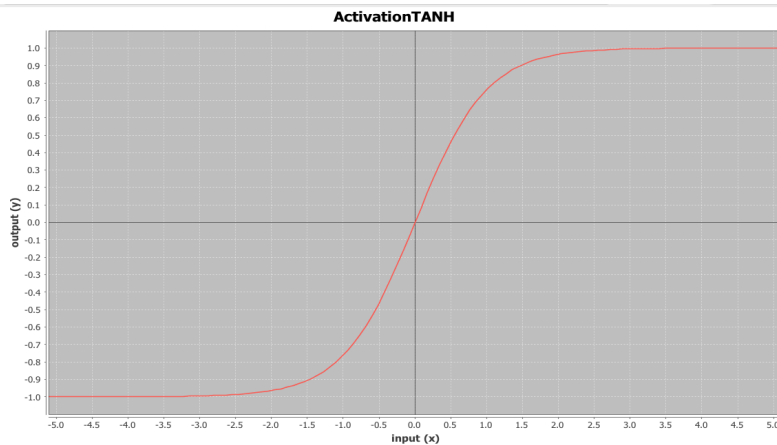


Abbildung 16: Die hyperbolische Tangentialfunktion als Aktivationsfunktion.

### 3.6.3.2 Layered Feed Forward Networks

Wir betrachten also Netzwerke von unseren Bausteile, das heisst also, wir verbinden sie zusammen. Wie sollen wir sie verbinden, oder besser gesagt, wie werden die Gewichte bestimmt? Eine Möglichkeit ist, eine Matrix-Inversion durchzuführen, und damit einen Gleichungssystem lösen. Für sehr grosse, überdefinierte Datenmengen ist diese Möglichkeit aber nicht realistisch. Deshalb nehmen wir einen iterativen Prozess in Angriff, wobei die Gewichte aus einem Training Set gelernt werden.

Im folgenden fokussieren wir auf einer bestimmten Netzwerk-Architektur, die auch die meistbenutzte ist, nämlich die sogenannte Layered Feed-Forward Networks. In solche Anordnungen sind also die Neuronen in Schichten angeordnet. Die erste Schicht ist die Input Schicht, die letzte ist die Output Schicht. Die Bedeutung von „Feed Forward“ kommt daraus, dass die Verbindungen nur nach Vorwärts gehen, also immer Richtung Output gehen.

Es ist bemerkenswert, dass die Netzwerke nicht unbedingt in diese Beschränkungen haben. Alle Anordnungen sind möglich. In weiteren Topologien wie die sogenannte Recurrent Networks, sowie die Self Organizing Maps können Verbindungen auch Rückwärts gehen. Wir werden solche Netzwerke aber hier nicht weiter betrachten.

Layered Feed Forward Networks werden auch Perceptrons genannt. Der einfachste Perceptron, auch Simple Perceptron genannt, besteht auch nur 2 Schichten, eine Input und eine Output-Schicht, wie in Abbildung 17 ersichtlich. In dieser Abbildung sieht man einen Netzwerk mit 4 Inputs (Grüne Neuronen), ein Output (Rot), und 1 gelbe Neuron. Das gelbe Neuron ist „always on“ Neuron genannt. Er hat die Eigenschaft, dass er immer feuert, unabhängig von Eingangswerte. Diese Eigenschaft wird später bei der Betrachtung der Lernalgorithmen benutzt.

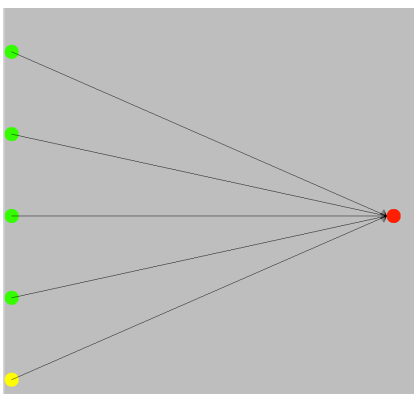


Abbildung 17: ein einfacher Perceptron (simple perceptron)

### 3.6.3.3 Eine einfache Lernmethode für Simple Perceptrons

Einmal das Netzwerk aufgebaut, muss es dann trainiert werden. Interessanterweise können sich neuronale Netzwerke für beide bewachtes oder unbewachtes Lernen eignen. Hier fokussieren wir auf bewachtes Lernen. (Eine Anwendung von neuronalen Netzwerke für unbewachtes Lernen wäre z.B. die Self Organizing Maps.)

Wie in den Abschnitten 3.6.1 und 3.6.2 betrachten wir einen Training Set. Dieser besitzt eine Menge Inputs, sowie eine Menge Outputs. Es muss nicht nur einen Output geben, da die Output-Schicht beliebig viele Neuronen haben kann. Wir können aber merken, dass fürs Lernen kann das Problem von mehrere Outputs immer als eine Zusammensetzung von mehrere Training Sets mit gleichen Input Datensätze und einen einzigen Output Wert.

Hier wird zuerst eine Lernmethode angegeben, die nur für eine einzige Schicht Neuronen (also: Input und Output-Schicht) gültig ist. Wir werden diese Methode dann für Multilayer Perceptrons später verallgemeinern.

Wir betrachten eine Schicht mit  $n$  Input-Neuronen. Die aufgeführte Lernmethode ist für alle Output-Neuronen gleich, deshalb fokussieren wir auf nur einen einzigen Output-Neuron.

Definitionsmässig besteht unser Training Set aus eine (beliebig grosse) Menge von  $n$ -dimensionalen Vektoren  $\vec{x} = \langle x_0, \dots, x_{n-1} \rangle$  sowie aus den erwarteten Output-Wert  $d$ . Es sind  $m$  Datensätze im Training Set vorhanden, so bezeichnen wir die Datensätze des Training Sets mit  $\vec{x}_j, j = 0 \dots m-1$  (Input-Werte), und  $d_j, j = 0 \dots m-1$  (Output-Werte). Damit werden einzelne  $x$ -Werte mit zwei Indizes gekennzeichnet,  $x_{i,j}$ . Diejenige Indizes für die Neuronen werden mit  $i$  und diejenigen für die Training-Datensätze mit  $j$  gekennzeichnet. Mit jedem Input-Neuron ist ein Gewicht assoziiert,  $w_i$ . In diese Gewichte, ist die Information des Netzwerkes tatsächlich enthalten. Um das Bias zu berücksichtigen fügen wir einen weiteren Neuron bzw. Gewicht  $w_n$ , damit haben wir  $n+1$  Gewichte, davon das Letzte  $w_n$  von keine Input-Wert abhängig ist. Das entspricht der Gelbe Neuron in Abbildung 17.

Das Lernalgorithmus sieht wie folgt aus:

1. Setze alle Gewichte auf einen willkürlichen Initialwert;
2. Für jede  $j = 0 \dots m-1$  :
  - a. Berechne  $y = g(w_0 x_{0,j} + w_1 x_{1,j} + \dots + w_{n-1} x_{n-1,j} + w_n)$
  - b. Berechne die Differenz zwischen  $y$  und den erwarteten Wert ;
  - c. Adaptiere die Gewichte :  $w_i := w_i + \alpha (d_j - y) x_{i,j}, i = 0, \dots, n-1$
  - d. Gehe wieder zu (a) bis entweder eine bestimmte Anzahl Iterationen vorbei sind oder der Fehler unter einen (Benutzer-)bestimmten Wert ankommt.

$y = g(\vec{x})$  ist die schon eingeführte Aktivationsfunktion, und  $0 \leq \alpha \leq 1$  ist der Lernfaktor.

### 3.6.3.4 Multistufige Perceptrons

Perceptrons mit nur Input- und Output-Schichten sind in sich nicht so interessant, weil ihre Fähigkeiten, „Patterns“ zu speichern, begrenzt sind. Es wurde gezeigt, dass das addieren von weitere Schichten die Speichermöglichkeiten erheblich erhöht. Damit sind neuronale Netzwerke vor allem interessant, wenn sie noch Zwischenschichten besitzen. Wir nennen diese Multi Layered Perceptrons. Abbildung 18 zeigt einen Beispiel für 4 Inputs, 3 Outputs und eine mittlere Schicht, auch versteckte Schicht oder Hidden Layer, mit 10 Neuronen. Ersichtlich sind auch die „always on“ Neuronen in Gelb. Dieses Beispiel ist auch wieder als feed-forward Netzwerk erkennbar, weil es nur Verbindungen, die nach Vorwärts gehen, vorweist.

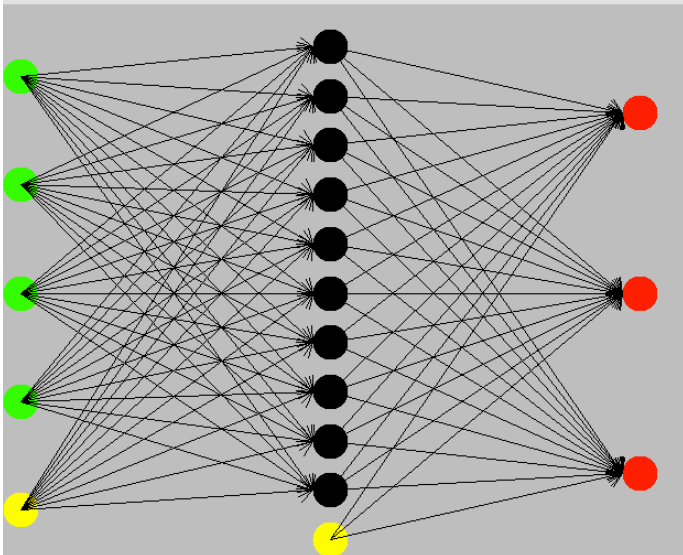


Abbildung 18: Ein multistufige Perceptron

Zwei Entscheidungen müssen für multistufige Netzwerke getroffen werden:

1. Wie viele Schichten muss unser Netzwerk besitzen und
2. Wie viele Neuronen sind in diese Schichten notwendig?

Probleme, die zwei Schichten benötigen, sind selten. Solche Netzwerke können alle Funktionen approximieren. Funktionen mit einer Schicht sind im allgemeinen genügend (siehe Tabelle). Es wurde gezeigt, dass mehr als zwei Schichten unnötig ist. In der Praxis kann man auch mit einer Schicht meistens umgehen.

Wie wählt man die Anzahl Neuronen in der versteckten Schichten? Auch wenn sie nicht direkt mit der externen Welt interagieren, haben sie einen starken Einfluss auf den Ausgangswerte. Wenn zu wenig Neuronen benutzt werden, ist das Resultat untergelöst (underfitted). In diesem Fall werden Patterns in einer komplizierten Datenmenge nicht genügend aufgelöst. Falls es zu viele Neuronen gibt, wird das Resultat übergelöst (overfitted), das heisst, für einfache Datenmengen mit begrenzten Information können die Neuronen nicht genügend trainiert werden, um das richtige Resultat zu finden. Ein weiteres Problem ist, dass das Netzwerk viel Zeit braucht, bis es trainiert ist (einen Lernvorgang kann Tage dauern), im schlimmsten Fall kann es unendlich viel Zeit brauchen, bis das Netzwerk trainiert ist.

Die Kunst mit neuronalen Netzwerke liegt also daran, das richtige Kompromiss zu finden. Das ist eine der grössten Schwäche dieser Methode – die Topologie des Netzwerk wirkt oft „gebastelt“, wobei die richtige Anzahl Neuronen mit Trial-and-Error ausgewählt werden. Das Netzwerk wird als Black Box angesehen, wo nichts berührt sein kann nach der Aufsetzung. Es ist aber zu merken, dass es Methoden gibt, um das optimale Netzwerk zu finden, diese werden aber nicht eingegangen.

### 3.6.3.5 Backpropagation Algorithmus

Da wir jetzt eine oder zwei versteckte Schichten haben, können wir unser einfachen Algorithmus aus Abschnitt 3.6.3.3 nicht mehr benutzen: wie könnten wir die Gewichte zwischen Input Schicht und versteckte Schicht aktualisieren? Tatsächlich soll eine Methode für Multischicht-Netzwerke ein Vorgehen besitzen, das die Fehler, die wir mit  $\Delta$  berechnet haben, von der Output zur Inputschicht rückpropagiert. Das meist bekannte (aber nicht mehr meistbenutzte) dieser Methoden ist Backpropagation genannt.

Der Backpropagation-Algorithmus kann für ein einzelnen versteckten Schicht in vier Schritte unterteilt werden :

1. Feed-Forward Berechnung: hier wird ein Datensatz aus dem Training Set dem Netzwerk präsentiert, und die Output Werte berechnet, sowie die Fehler gegenüber den erwarteten Resultate. Das entspricht dem einfachen Algorithmus vom Abschnitt 3.6.3.3.
2. Der Fehler wird aus dem Output-Neuronen zu den mittleren Neuronen rückpropagiert, in dem man aufgrund der Fehler der Output Schicht die Fehlern in der mittleren Schicht berechnet.
3. Der Fehler wird aus den mittleren Neuronen zu dem Input-Schicht rückpropagiert, in dem man aufgrund der Fehler der mittlern Schicht die Fehlern in der Output Schicht berechnet.
4. Man benutzt diese Fehler, um die Werte der Gewichte neu zu berechnen.

Der Algorithmus stoppt, wenn der Fehler genügend klein wird.