6. Mehrschichten-Netze (Multilayer-Perzeptrons MLP)

6.1 Einleitung

In den letzten Aufgabenblättern haben wir uns Netzwerke angesehen, die als Beispiel für ein erstes einfaches Netze aus formalen Neuronen gelten können. Eine natürliche Erweiterung davon sind die sogenannten Mehrschichten-Netzwerke. Sie bestehen aus mehreren Schichten von Neuronen, die man auch als Hintereinanderschaltung von *single layer*-Netzen auffassen kann. In Abbildung 6.1 ist eine allgemeine *Feedforward*-Architektur eines Netzes dargestellt.

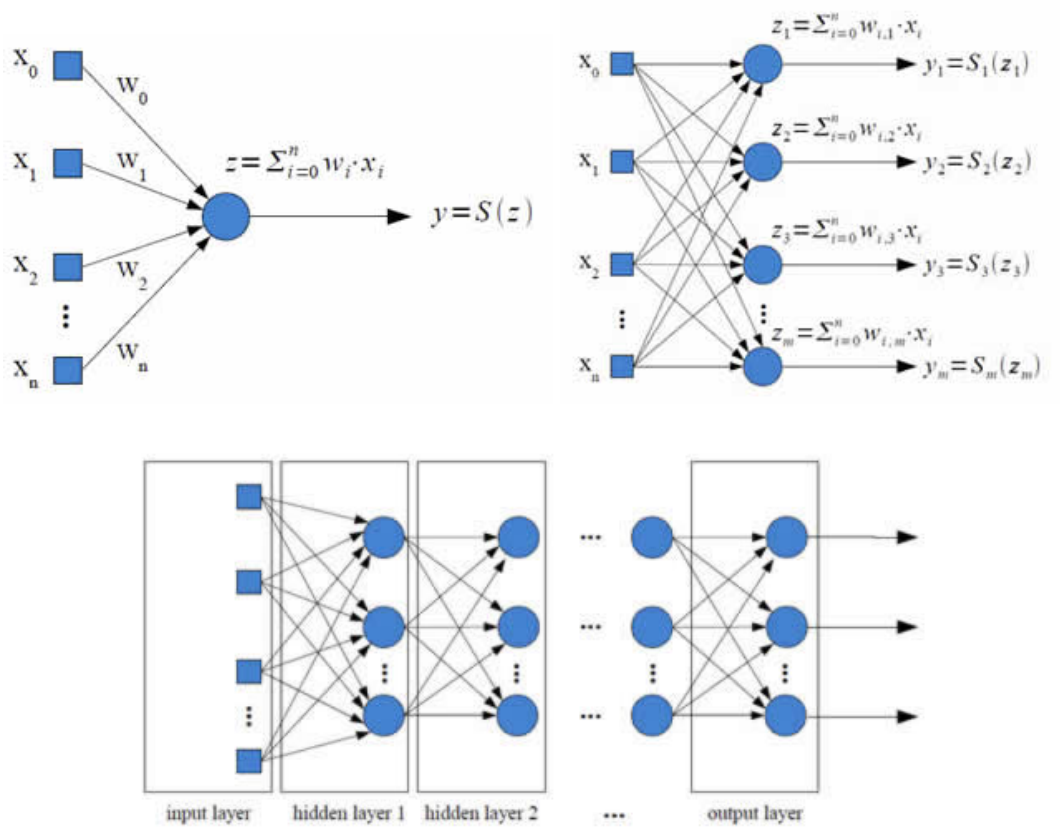


Abbildung 6.1 Darstellung eines formalen Neurons, einer Neuronenschicht und einer mehrschichtigen *Feedforward*-Architektur.

Im Prinzip handelt es sich dabei um ein verallgemeinertes Perzeptron, bei dem neben der Ausgabeschicht noch mehrere Zwischenschichten, sogenannte *hidden layer*, existieren können. Aus diesem Grund wird ein derartiges Netz auch als *Multilayer-Perzeptron* (MLP) bezeichnet.

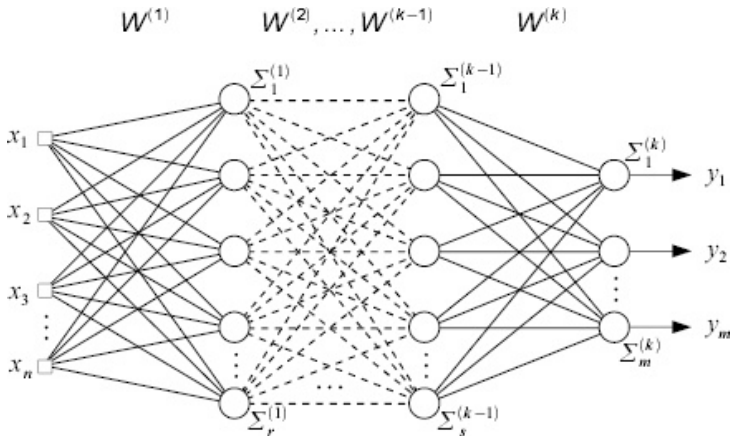


Abbildung 6.2 Darstellung eines Multilayer-Perzeptrons.

Die Aktivität und die Ausgabe eines solchen Netzes errechnet sich schrittweise, Schicht für Schicht. Für einen gegebene Zeitpunkt *t* wird mit der Eingabe **x** bei der ersten Schicht eine Ausgabe **y** erreicht. Für die *j*-te Schicht ist die skalare Ausgabe $y_i^{(j)}$ des *i*-ten Neurons hier wie üblich gebildet durch $y_i^{(j)} = S_j\left(\underline{y}^{(j-1)^T}\underline{w}_i^{(j)}\right)$ mit Hilfe der gewichteten Summe $z = \mathbf{w}^T\mathbf{y}$ und der Ausgabefunktion S(z). Dabei bezeichnet $\underline{y}^{(j-1)}$ den Ausgabevektor der vorigen Schicht, der als Eingabevektor der *j*-ten Schicht dient. Wir treffen der Einfachheit halber die Konvention, daß $\underline{y}^{(0)} = \underline{x}$ gilt für die Eingabe $\underline{x}$ des Netzes. $\underline{w}_i^{(j)}$ ist der Gewichtsvektor des i-ten Neurons in Schicht *j* und $S_j(\cdot)$ eine beliebige *differenzierbare* sigmoidale Ausgabefunktion für alle Neuronen dieser *j*-ten Schicht, etwa die Fermifunktion oder der Tangens-Hyperbolicus, siehe Abbildung 2.

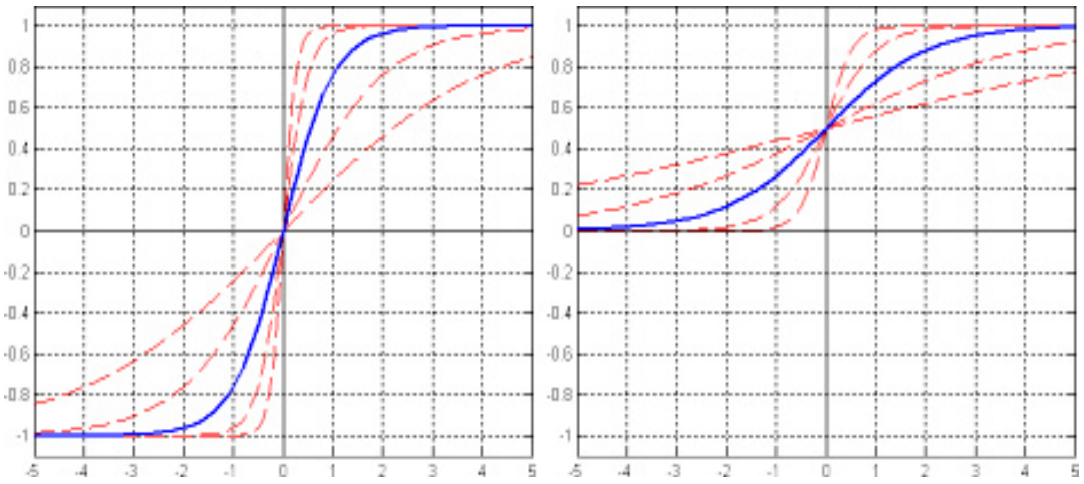


Abbildung 6.3 Übliche Ausgabefunktionen.

Links: Tangens Hyperbolicus $S(z) = \tanh(z \cdot a) = \frac{1-\exp(-2z \cdot a)}{1+\exp(-2z \cdot a)}$

Rechts: Fermifunktion $S(z) = \frac{1}{1+\exp(-z \cdot a)}$

Für $a = 1$ ergeben sich die blauen Kurven. Rot gestrichelt sind die Ausgabefunktionen für $a = 4, 2, \frac{1}{2}, \frac{1}{4}$ dargestellt.

Fassen wir mit $W^{(j)} = [\underline{w}_1^{(j)}, ..., \underline{w}_l^{(j)}]$ die Gewichtsvektoren der *j*-ten Schicht zu einer Matrix zusammen, dann berechnet sich der Ausgabevektor $\underline{y}^{(j)} = \left[y_1^{(j)}, ..., y_l^{(j)}\right]^T$ der Neuronen dieser Schicht als:

$$\underline{y}^{(j)} = S_j\left(W^{(j)^T} \cdot \underline{y}^{(j-1)}\right)$$

Folglich besitzt die Ausgabe $\underline{y}^{(k)}$ des gesamten Netzes bei Eingabe eines $\underline{x} \in \mathbb{R}^n$ die Form:

$$\underline{y}^{(k)} = S_k\left(W^{(k)^T} \cdot ... \cdot S_2\left(W^{(2)^T} \cdot S_1\left(W^{(1)^T} \cdot \underline{x}\right)\right)\right)$$

Es hat sich herausgestellt, daß ein solches Netz, bestehend aus *k=2* Schichten (eine *hidden layer*-Schicht und eine Ausgabeschicht) von informationsverarbeitenden Neuronen, bereits ausreicht, um beliebige kontinuierliche Funktionen beliebig genau approximieren zu können! Dabei wird vorausgesetzt, daß die Neuronen des *hidden layers* sigmoidale Ausgabefunktionen und die Neuronen der Ausgabeschicht lineare Ausgabefunktionen besitzen.

Man kann außerdem zeigen, daß sich mit einem Netz, bestehend aus *drei* Schichten (also zwei *hidden layern*), *jede beliebige* Funktion beliebig genau approximieren läßt. Multilayer-Perzeptrons sind also außerordentlich ausdrucksstark. Aus praktischer Sicht ist daher ein Netz mit einem einzigen *hidden layer* in den meisten Fällen bereits ausreichend.

Allerdings bedeuten diese Aussagen nur, daß es eine Gewichtseinstellung für die Neuronen des betreffenden Netzes *gibt*. Wie man dem Netz diese Gewichte antrainiert, ist dabei eine andere Frage, der wir im Folgenden nachgehen wollen.

nach oben

6.2 Training mit Backpropagation: Fehlerrückführung in Mehrschichten-Netzen

Für derartige Netze existieren sehr viele verschiedene Trainingsalgorithmen. Der populärste unter ihnen ist dabei sicherlich der sogenannte "Backpropagation-Algorithmus". Für einen Eingabevektor **x** und einen anzutrainierenden Ausgabevektor $\ell(x) \in \mathbb{R}^m$ gliedert er sich in drei Phasen, die in Abbildung 3 dargestellt sind. In der ersten Phase wird die Ausgabe $\mathbf{y}^{(2)}(t)$ errechnet und in der zweiten Phase der Fehler (Differenz zwischen gewünschter und tatsächlicher Ausgabe) in den verschiedenen Schichten, beginnend mit der letzten Schicht und endend in der ersten. In der dritten Phase werden schließlich die Gewichte in den verschiedenen Schichten korrigiert.

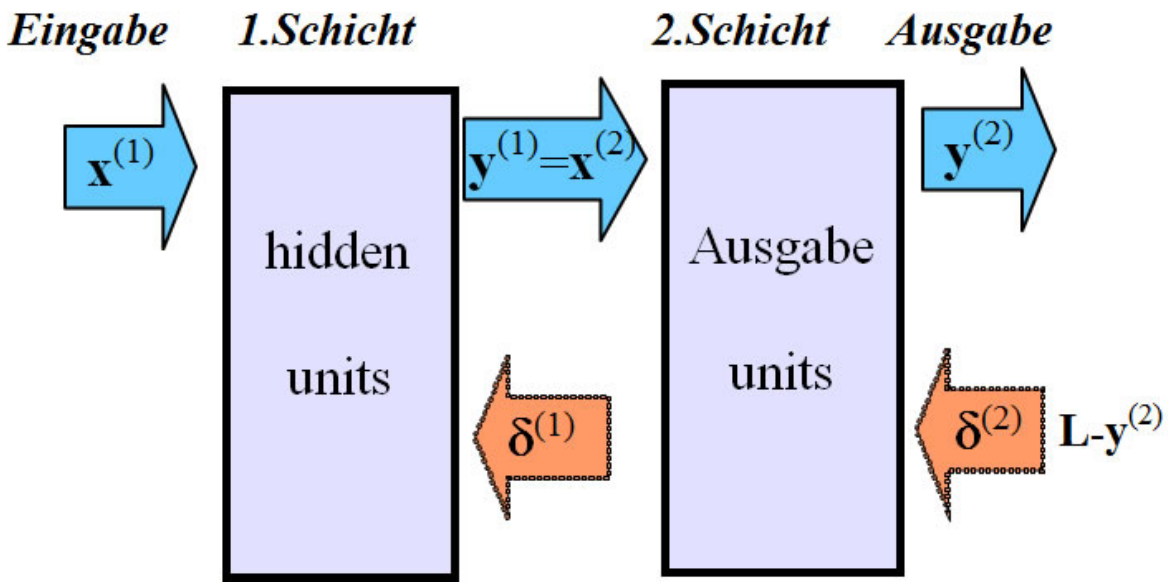
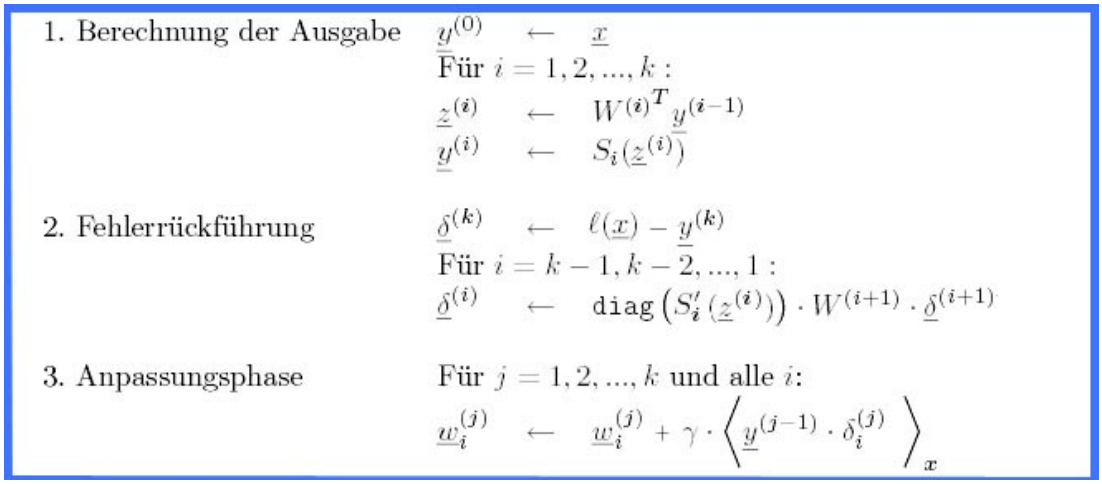


Abbildung 6.4 Darstellung eines Multilayer-Perzeptrons, einer mehrschichtigen *Feedforward*-Architektur.

Der Algorithmus implementiert dabei einen stochastischen Gradientenabstieg im Raum aller Gewichte und sucht dort nach einem lokalen Minimum bezüglich des erwarteten quadratischen Fehlers zwischen der Netzausgabe und der anzutrainierenden Ausgabe. Die Gewichtsmatrizen $W^{(1)}, ..., W^{(k)}$ wollen wir so einstellen, daß unser Netz bei Eingabe eines Vektors $\underline{x}$ einen möglichst geringen Fehler zu einem vorgegebenen Ausgabevektor $\ell(\underline{x}) \in \mathbb{R}^m$ produziert. Als Optimierungsziel nehmen wir uns daher vor, den *erwarteten quadratischen Fehler* (in Form des Abstandsquadrats) zwischen den Eingabevektoren und den geforderten Ausgabevektoren zu minimieren.

Damit erhalten wir zusammenfassend die folgenden Lernregeln, für ein Netz, bestehend aus *k* Schichten von Neuronen, wobei jede Schicht eine eigene sigmoidale Ausgabefunktion $S(\cdot)$ besitzen kann:



Aufgrund von Schritt 2 wird diese Lernregel auch als *backpropagation of error* oder einfach als *Backpropagation*-Lernregel bezeichnet, da der Ausdruck $\underline{\delta}^{(k)} = \underline{y}^{(k)} - \ell(\underline{x})$ gerade den negativen Fehlervektor der Netzausgabe darstellt und die $\underline{\delta}^{(i)}$ gewissermaßen ein "durch die Schichten zurückgeführtes" Fehlersignal darstellen. Die kleine Konstante gamma > 0 gibt die Schrittweite des Gradientenverfahrens an und wird in diesem Zusammenhang auch als Lernrate des Algorithmus bezeichnet. Üblicherweise wählt man sie in der Größenordnung ca. 0,1 - 0,001. Die Lernregel mit den Erwartungswertklammern stellt die sogenannte *Batch*- oder *Offline*-Variante dar.

Die Erwartungswertklammern in Schritt 3, der Anpassungsphase, können auch weggelassen werden. Man erhält dann eine *stochastische Approximation* der Lernregel und ist nicht mehr auf eine Summation über mehrere Eingabemuster $\underline{x}$ angewiesen, um den Gradienten zu berechnen, sondern kann nach jedem einzelnen Muster einen Lernschritt ausführen. Aus diesem Grund wird diese Variante dann auch als *Online-Verfahren* der Lernregel bezeichnet.

nach oben

6.3 Testen von Mehrschichten-Netzen

Die Probleme dieses Aufgabenblattes sind etwas knifflig, so dass es hier ganz besonders wichtig ist, auf ein korrekt funktionierendes Programm vertrauen zu können. Ganz allgemein können wir die Korrektheit eines Verfahrens nicht aus seiner Funktion auf unbekannten Daten allein herleiten, so dass wir spätestens an dieser Stelle des Praktikums eigene Testwerkzeuge

benötigen.

Für die Korrektheit des Programms benötigen wir zum einen Tests mit sehr einfachen Trainingsmustern, die vorhersehbare Ergebnisse liefern müssen. Zur Kontrolle des Konvergenzverhaltens der einzelnen Netzeinheiten (wie Neuronen oder Schichten) müssen wir ausserdem den zeitlichen Verlauf des Fehlers (Wert der Zielfunktion zum Zeitpunkt *t*) und die Länge des Gradienten (zum Zeitpunkt *t*) während des Trainings genau beobachten. Das Verhalten des Gradienten und des Fehlers lässt Rückschlüsse darauf zu, wie die Lernparameter (Lernrate, Wertebereich der Gewichtsinitialisierung, Neuronenzahl, etc.) anzupassen sind und ob der Lernalgorithmus korrekt implementiert wurde.

Bei diesen Netzen unterteilen wir in zwei Testphasen: zum einen das Testen der Aktivität, zum anderen das Testen der Fehlerverbesserung.

Die **erste Phase** beginnt mit der Definition eines sehr einfachen Testmusters und einer dazu passenden Belegung der Gewichte. Beides wird so gewählt, dass die Ausgabe klar vorhergesagt werden kann. In Python können wir für jede Schicht im Debugging-Modus die Aktivitäten per Hand festlegen und damit vorhersagbare Ergebnisse produzieren.

> ***Beispiel**: Belegen wir alle Eingaben eines Neurons mit 1 und seine Gewichte ebenfalls mit 1, dann erhalten wir bei der Fermi-Ausgabefunktion gerade die Ausgabe 1. Wählen wir dagegen für alle Gewichte 0, so muss sich 0,5 als Ausgabe ergeben.*

Die **zweite Phase** besteht in der Ausgabe des Fehlers und dem Nachweis, dass der erzielte Fehler während des Trainings geringer wird. Dazu wird bei Eingabe eines einzelnen Musters nach Bildung der Aktivität der Fehler des Netzes ermittelt. Nun werden die Gewichte mit Hilfe der Lernregel verbessert und der Fehler neu ermittelt. Der Fehler sollte sich vermindert haben, sonst haben wir falsch programmiert. Dies können wir für jede Schicht durchführen, wobei wir immer das selbe Muster als Eingabe für unser Netz verwenden.

Zum Abschluss beobachten wir das **dynamische Verhalten** des Netzes beim Lernen über mehrere Muster, indem wir uns den zeitlichen Verlauf der Gradientenlänge und des durchschnittlichen Fehlers ausgeben lassen. Der Fehler des Netzes wird dabei separat auf jeweils drei verschiedenen Mengen von Mustern gemessen:

- zum einen auf der zum Training verwendeten **Trainingsmenge**,
- zum anderen auf einer zusätzlichen **Validierungsmenge**, anhand der während des Trainings der Fehler des Netzes beurteilt wird (zum Beispiel für *stopped training*),
- sowie einer separaten (nicht im Trainingsprozess verwendeten) **Testmenge** zum

objektiven Überprüfen der Generalisierungsfähigkeit nach Abschluss des Trainings.

Da der Backpropagation-Algorithmus einen Gradientenabstieg implementiert, um ein lokales Minimum der Zielfunktion zu finden (dort stellt der Gradient den Nullvektor dar), sollte die Länge des Gradienten bei Konvergenz des Verfahrens gegen Null gehen. Aus den gleichen Gründen sollte der Fehler während des Trainings insgesamt abnehmen. Ist dies nicht der Fall, können die Trainingsparameter falsch gewählt sein, oder unser Programm enthält Fehler. Letzteres können wir in der ersten Phase überprüfen.

nach oben

6.4 Einige Anmerkungen zu Backpropagation

Wir sehen jetzt auch, warum die Forderung an die Differenzierbarkeit der Ausgabefunktion $S(\cdot)$ notwendig war, da deren Ableitung in der Lernregel verwendet wird. In den meisten Fällen wird man die Fermifunktion oder den Tangens-Hyperbolicus als nichtlineare Ausgabefunktion der Neuronen verwenden. Für letzteren gibt es bestimmte analytische Gründe, ihn als bevorzugte Funktion in den Hiddenlayern von MLPs einzusetzen. In der letzten Schicht des Netzes (außer bei Klassifikationsproblemen) wird in der Regel die lineare Ausgabefunktion $S(z) = z$ mit der Ableitung $S'(z) = 1$ verwendet.

Allerdings gibt es noch eine wichtige Eigenschaft in Zusammenhang mit sigmoidalen Ausgabefunktion wie der Fermi- oder der `tanh`-Funktion zu beachten. Betrachten wir uns dazu Abbildung 3.

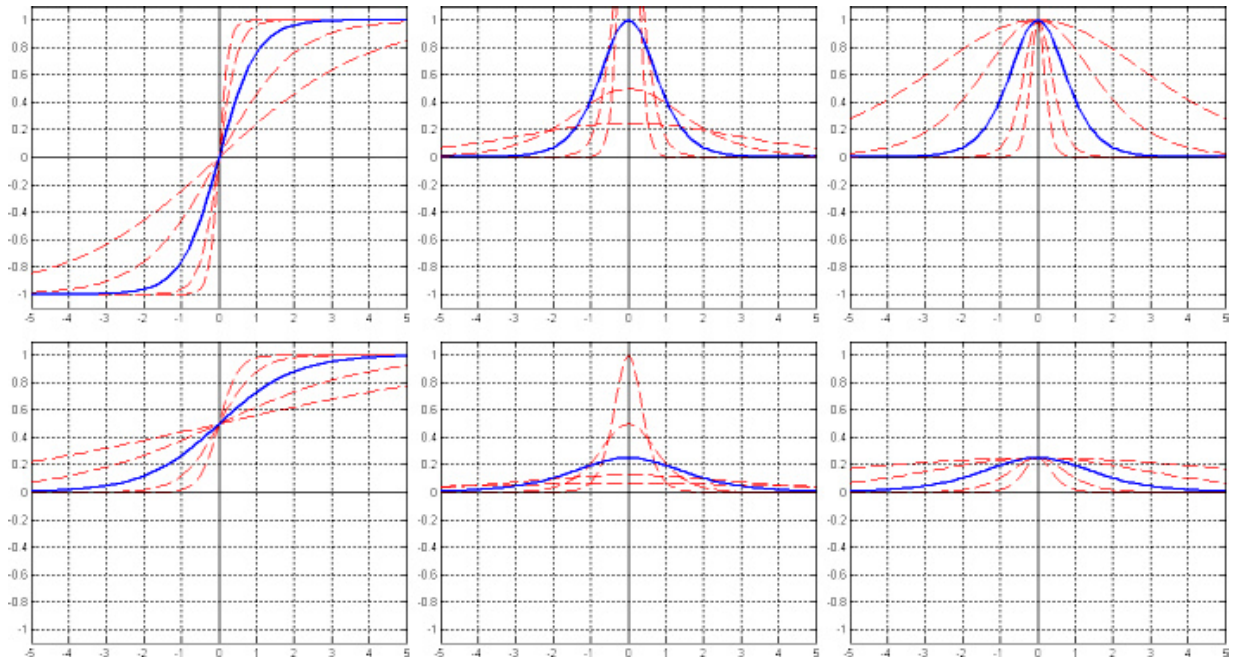


Abbildung 6.5 Ableitung der tanh- und der Fermifunktion für verschiedene Neigungsparameter $a = 4, 2, 1, \frac{1}{2}, \frac{1}{4}$.

	Links	Mitte	Rechts
Oben	Tangens-Hyperbolicus $S(z) = \tanh(z \cdot a)$	Ableitung $S'(z) = a \cdot (1 - S(z)^2)$	Ableitung ohne Vorfaktor $\frac{1}{a} \cdot S'(z) = (1 - S(z)^2)$
Unten	Fermifunktion $S(z) = \frac{1}{1+\exp(-z \cdot a)}$	Ableitung $S'(z) = a \cdot (1 - S(z)) \cdot S(z)$	Ableitung ohne Vorfaktor $\frac{1}{a} \cdot S'(z) = (1 - S(z)) \cdot S(z)$

Dargestellt sind die Ableitungen dieser Funktionen. Wie an den Graphiken zu erkennen ist, haben die Funktionswerte der Ableitungen die Eigenschaft, sich für betragsmäßig große Argumente immer weiter dem Wert Null anzunähern. Problematisch daran ist, daß der Gradient in unserer Lernregel diese Ableitungen als Vorfaktoren besitzt, was bedeutet, daß der Gradient bei betragsmäßig großer Aktivität der Neuronen ebenfalls gegen Null geht. Das hat zur Folge, daß unser Netz jedoch nichts mehr lernen wird!

Wie in der Abbildung zu erkennen, kann dies jedoch mit einer geeigneten Wahl des Neigungsparameters *a* kompensiert werden. Allerdings sollte man beim Bilden der Ableitung den Neigungsparameter als Vorfaktor der Ableitung entfernen. Der Grund liegt darin, daß er sonst ungewollten Einfluß auf die Lerngeschwindigkeit nehmen kann. An der dritten Spalte aus Abbildung 4 ist gut zu erkennen, dass bei Entfernen von *a* aus der Ableitung die Öffnung der

Kurve - und damit der Wertebereich, in dem noch signifikant gelernt werden kann - zunimmt, ohne daß sich der maximale Funktionswert verändert. Im Gegensatz dazu beeinflussen verschiedene Werte von *a* in der ursprünglichen Ableitung die Lerngeschwindigkeit drastisch, da sich hier die maximalen Funktionswerte stark verändern.

Eine weitere Idee besteht daraus, die Gewichte der ersten Schicht so optimal zu gestalten, dass sie den quadratischen Fehler minimieren, der für die Aktivität z (den linearen Fall) anfällt. Dies bedeutet, einfach die Eigenvektoren der Kovarianzmatrix $\mathbf{C}_{xx}$ der Eingabe **x** zu bilden mit den größten Eigenwerten (d.h. eine PCA durchzuführen), und dann die diese Eigenvektoren als Gewichtsvektoren der *hidden*-Schicht zu verwenden. Starten wir mit diesen Gewichtswerten, so sind nur noch geringe Korrekturen beim Training fällig, um mit den nicht-linearen Ausgabefunktionen zu einem lokalen Minimum zu kommen.

Backpropagation gehört mit zu den populärsten Lernregeln in Zusammenhang mit Neuronalen Netzen. Es war die erste Lernregel mit der Fähigkeit, die Gewichte eines Netzes mit einer derart großen Ausdruckskraft einlernen zu können. Allerdings hat die Sache einen Harken, denn der Gradientenabstieg, den wir hier zum Lernen verwenden, kann unter Umständen sehr viele Iterationen benötigen, bis ein Minimum der Fehlerfunktion gefunden wurde. Daher gibt es inzwischen auch eine ganze Reihe von Verbesserungen (zu nennen sind etwa QuickProp oder RProp) allerdings sind diese Algorithmen auch dementsprechend komplexer. Desweiteren läuft ein Gradientenabstieg immer Gefahr, lediglich gegen ein *lokales* Minimum zu konvergieren, so daß womöglich keine bestmögliche Einstellung der Gewichte gefunden wird. Im Gegensatz zu Adaline gibt es bei sämtlichen Trainingsalgorithmen für Multilayer-Perzeptrons leider *keine* Garantie dafür, daß die Gewichte gegen ein globales Minimum konvergieren.

Eine wirklich bemerkenswerte Tatsache ist allerdings, daß man in der Praxis häufig beobachten kann, daß dies eigentlich kaum eine Rolle zu spielen scheint! Die Einzelheiten sind hier allerdings noch nicht hundertprozentig geklärt. Im übrigen scheint die Online-Variante von Backpropagation auch etwas weniger Gefahr zu laufen, in einem lokalen Minimum gefangen zu werden, als die Offline (Batch)-Version. Man kann auch recht einfach zeigen, daß die Gewichte der *letzten* Schicht durch Backpropagation gegen eine optimale Anpassung bezüglich der übrigen Gewichte streben, wenn man eine lineare Ausgabefunktion für die letzte Schicht verwendet (der Beweis funktioniert dann wie im Fall von Adaline).

Ein wichtiger Punkt, den es beim Training mit Backpropagation noch zu beachten gibt, ist der, dass man beim Training keine feste oder vorhersehbare Reihenfolge der Eingabemuster (Eingabevektoren) verwenden sollte. Interessanterweise ist ein Multilayer-Perzeptron in der

Lage, sich nach ein paar Trainingsdurchläufen mit der gleichen Menge an die Reihenfolge der präsentierten Ausgabevektoren anzupassen. Daher empfielt es sich, zu Anfang jeder Trainingsepoche die Reihenfolge der Eingaben zufällig zu permutieren und dann erst dem Netz zum Training anzubieten. Die Leistung des Netzes kann dadurch in vielen Fällen noch etwas verbessert werden.

nach oben