Neuronale Netze und Tiefe Architekturen

Christian Wurm
cwurm@phil.hhu.de

January 9, 2018

Abstract

Grundlagen für die Theorie des Deep Learning, wie es momentan benutzt wird. Manuskript zum Gleichnamigen Seminar mit Younes Samih, WS17/18 an der HHU. Kommentare willkommen!

Contents

1 Maschinelles Lernen: Klassifikation und Regression

Was man im maschinellen Lernen "lernen" oder besser gesagt *approximieren* will sind normalerweise Funktionen. Da der Begriff sehr allgemein ist, lässt sich normalerweise alles darunter packen. Man unterscheidet aber zwei wichtige Fälle:

1. Wir approximieren eine Funktion $f : X \to M$, wobei M eine endliche Menge ist;

2. wir approximieren eine Funktion $f : X \to \mathbb{R}$, wobei $\mathbb{R}$ die reellen Zahlen sind.

Hier gibt es folgendes anzumerken: natürlich gibt es auch andere logische Möglichkeiten, aber die treten normalerweise nicht auf bzw. lassen sich reduzieren. Es spielt auch keine Rolle, ob die Ausgangsmenge X endlich oder unendlich ist; sie ist nur normalerweise wesentlich größer als unsere Datensätze – sonst gäbe es ja keinen Grund ML zu verwenden.

Im Fall 1 spricht man von **Klassifikation**; man sagt also die Funktion f *klassifiziert* Gegenstände in X als Gegenstände in M. Ein klassischer Fall wäre Objekterkennung; man nimmt ein Pixel.-Bild (das sich als eine Matrix darstellen lässt) als Eingabe, und gibt ein label zurück, dass besagen soll welches Objekt dargestellt wird.

Im Fall 2 spricht man von **Regression**; wir haben hier üblicherweise $X = \mathbb{R}$, und es geht darum, aus einer gewissen (endlichen) Menge von Koordinaten eine Funktion zu extrapolieren, die durch diese Koordinaten geht (oder zumindest in der Nähe der Koordinaten passiert).

Der zentrale Punkt ist dabei: es gibt viele Möglichkeiten das zu tun, und es ist schwer zu sagen dass eine Lösung besser ist als eine andere. Intuitiv wäre die Antwort: die einfachste. Das ist aber nur in manchen Fällen ein sinnvolles Kriterium, weil es viele verschiedene Definition von einfach gibt. D.h. die Antwort lautet: ob eine Approximation gut ist oder nicht, ist letztenendes eine empirische Frage.

Welche Methode am besten funktioniert, ist durchaus vom Problem abhängig: es kann sein dass eine Methode für ein Problem sehr gut funktioniert, für ein anderes Problem nicht. Allerdings haben die Methoden, die wir hier behandeln wollen, in den letzten Jahren auf praktisch allen Feldern der künstlichen Intelligenz durchschlagende Erfolge erzielt. Warum das so ist weiß eigentlich niemand genau.

1.1 Der Rahmen

Klassifikation ist ein erstes Beispiel für induktive Inferenz. Was dabei induziert wird ist eine

$$\text{Funktion } f, \text{ (z.B. } F : M \to N)$$

und zwar eine diskrete Funktion, d.h. eine Funktion die nur endlich viele verschiedene Eingaben nimmt und damit nur endliche viele Ausgaben liefert. Das Klassifikationsproblem ist also folgendes:

> Gegeben eine endliche Teilmenge von Instanzen von f, liefere eine Funktion h die f *approximiert*.

Wir sagen, dass $(m, n) \in M \times N$ eine **Instanz** von f ist, falls $f(m) = n$.

- Falls f eine stetige Funktion ist (z.B. $f : \mathbb{R} \to \mathbb{R}$), dann spricht man von *Regression*,
- falls f nur endlich viele Eingaben (und damit Ausgaben) hat, spricht man von **Klassifikation**.

Wir nennen wir unsere **Hypothese**.

$$h, \text{ wobei } h : M \to N$$

Das Grundproblem ist dass wir f normalerweise nicht kennen, d.h. wir können nie wissen, ob unsere Induktion erfolgreich war oder nicht. Alles was wir wissen können ist ob h übereinstimmt mit f auf dem endlichen Datensatz, den wir zur Verfügung haben. Eine entscheidende Rolle spielt dabei der sogenannte

Hypothesenraum H,

d.i. eine Menge von möglichen Funktionen, aus der wir h auswählen. Der Raum **H** ist durchaus nicht vorgegeben im Rahmen des Induktionsproblems, und die Wahl ist oft alles andere als einfach.

Das sieht man sehr schön am Beispiel einer *Regression*. Nehmen wir an, wir möchten eine Funktion

$$f : \mathbb{R} \to \mathbb{R}$$

induzieren, z.B. um die Korrelation von Tagestemparatur und Straftaten an einem gewissen Ort zu bestimmen (letztere gemittelt über einen längeren Zeitraum).

Was wir also gegeben haben ist eine Menge von Zahlenpaaren der Form

$$(10, 25.3),$$
$$(14, 27.8)$$
$$\text{etc.}$$

Nennen wir diese Menge D, unseren Datensatz. Da D eine Teilmenge des **Funktionsgraphen** von f ist, ist unsere Aufgabe ist wie folgt umrissen:

Finde eine Funktion $h \in H$, so dass für alle $(x, y) \in D$, $h(x) \approx y$.

Man nennt das auch die Konsistenzbedingung: die Hypothese soll konsistent mit den Daten sein. Und jetzt die Frage: was ist H? Hier gibt es folgende Überlegungen:

Je einfacher h ist, desto überzeugender würden wir es finden.

Z.B.: sei

$$(1) \quad h_1(x) = \frac{x}{2} + k$$

Das wäre sehr schön, und wir könnten sagen: ein Anstieg von 2° Celsius bedeutet eine zusätzliche Straftat. Wir könnten auch sagen: wenn es im

August im Schnitt 25° wärmer ist als im Dezember, dann haben wir im Schnitt 12.5 Straftaten mehr pro Tag. Das wäre also eine sehr interessante Entdeckung!

Andererseits, es ist sehr unwahrscheinlich dass ein so komplexer, mittelbarer Zusammenhang so einfach ist, und so ist es sehr unwahrscheinlich, dass für alle $(x, y) \in D$ wir tatsächlich $h_1(x) = y$ haben. Es gibt sicherlich eine Funktion h_2, die in dieser Hinsicht wesentlich besser ist, z.B.

$$h_2(x) = x^5 + 6x^4 - 14x^3 + 15x^2 - 8x \tag{2}$$

Nehmen wir an, h_2 ist genauer auf D als h_1. Würden Sie sagen, dass h_2 plausibler ist? Eher nicht: wir würden sagen, dass die Komplexität von h_2 ein Anzeichen dafür ist, dass sie "maßgeschneidert" ist auf D und

schlecht generalisiert.

Das liegt v.a. daran, dass h_2 extrem komplex ist im Vergleich zu h_1. Wir treffen hier auf ein sehr grundlegendes Prinzip, nämlich das sog. **Rasiermesser von Ockham** (*Ockham's razor*), das besagt:

> *Die beste Hypothese aus einer Anzahl von Hypothesen die konsistent sind mit den Daten ist die einfachste.*

Allerdings sieht man bereits an unserem Beispiel, dass das eine sehr weiche Bedingung ist: denn h_2 passt besser als h_1, und es hängt nun alles davon ab, wie wir Konsistenz definieren. Es handelt sich also um eine weiche Richtlinie (die nichtsdestotrotz von grundlegender Bedeutung ist).

Wir können dieses Problem evtl. vermeiden, indem wir unseren Hypothesenraum *a priori* beschränken. Z.B. können wir sagen: uns interessieren nur die Polynome 2ten Grades, also Funktionen der Form

$$x^2 + ax + b \tag{3}$$

Dabei gibt es folgendes zu beachten:

- Je kleiner der Hypothesenraum H, desto einfacher ist es, zwischen den konsistenten Hypothesen einen Kandidaten auszuwählen.
- Aber: je kleiner die Hypothesenraum, desto größer ist auch die Wahrscheinlichkeit, dass die korrekte Funktion gar nicht darin enthalten ist, also $f \notin H$.

Es gibt also Gründe die dafür und dagegen sprechen, H zu verkleinern. Wenn z.B. $f \notin H$, dann haben wir natürlich keine Möglichkeit, die korrekte Funktion zu induzieren. Da wir f nicht kennen, gibt es keine Möglichkeit, dass auszuschließen.

Nehmen wir z.B. an, die korrekte Korrelation (die wir natürlich nicht kennen) wäre

$$f(x) = ax + b + c\sin(x) \tag{4}$$

das bedeutet: wir haben eine wachsende Wellenfunktion: Kriminalität erlebt bei steigenden Temperaturen immer wieder Scheitelpunkte.

- Solange wir also annehmen dass H aus Polynomialen besteht, werden wir niemals die richtige Funktion finden, sondern immer unmöglichere Polynomialfunktionen suchen müssen, solange wir mit neuen Daten konfrontiert werden!

Wir sehen also wie wichtig der richtige Hypothesenraum ist!

2 Maschinelles Lernen – Überblick

2.1 Was brauchen wir?

Um erstmal einen Überblick zu bekommen ist es hilfreich, zu erklären was man braucht um einen ML-Algorithmus zum laufen zu bringen. Wir brauchen dazu:

1. Ein Modell, das wir trainieren sollen, und dass am Ende die Aufgabe beherrschen soll. "Modell" heißt im Prinzip soviel wie: Klasse von Funktionen.

2. Ein Datenformat, mit dem wir das Modell trainieren können. Es ist wichtig das dieses Format festgelegt ist.

3. Eine Kostenfunktion, die uns sagt, wie weit unser Modell "daneben-liegt" von der richtigen Vorhersage (Kosten $\cong$ Falschheit).

4. Eine Optimierungsprozedur, mittels derer wir das Modell schrittweise verbessern, bis wir konvergieren oder entscheiden dass wir genug "gelernt" haben.

Das ist bereits alles. Was wichtig ist zu verstehen ist dass diese Komponenten eng zusammenhängen. Z.B. nehmen wir einmal an, unser Eingaben machen den Raum $\mathbf{X}$ aus, unsere Ausgaben den Raum $\mathbf{Y}$. Wir können normalerweise annehmen, dass

$$\mathbf{X} \subseteq \mathbb{R}^n,\ \mathbf{Y} \subseteq \mathbb{R}.$$

Unsere Zielfunktion ist also eine Funktion

$$f : \mathbf{Y} \to \mathbf{X}$$

Nehmen wir an, unser Modell M ist parametrisiert durch eine Zahl/Vektor/Matrix $\theta \in \Theta$, dem Parameterraum. Das kann z.B. eine Zahl sein im Fall das wir den Mittelwert einer Normalverteilung suchen; wir nehmen mal diesen einfachsten Fall an. Wir haben also eine Kandidatenmenge $M(\theta) : \theta \in \Theta$, und wir suchen dasjenige θ, das am besten funktioniert. Nun nehmen wir unseren Datensatz $D \subseteq \mathbf{X} \times \mathbf{Y}$, also die Bebachtungen, die wir haben, und unsere Kostenfunktion $K : \mathbf{Y} \times \mathbf{Y} \to \mathbb{R}$. Damit können wir unser Problem etwas formaler auffassen: wir suchen dasjenige θ, so dass

$$\sum_{(x,y)\in D} K(D(x), M(\theta)(x))$$

minimal wird, wir suchen also

$$\underset{\theta\in\Theta}{argmin} \sum_{(x,y)\in D} K(D(x), M(\theta)(x)) \tag{5}$$

Das heißt, unsere Optimierung basiert eigentlich darauf, eine geschlossene analytische Funktion zu minimieren. Das ist zwar konzeptuell einfach, aber nicht unbedingt in der Praxis: wenn

$$L(D(x), M(\theta)(x)) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + b \tag{6}$$

die resultierende Funktion ein Polynom ist, wobei $\Theta \subseteq \mathbb{R}^n$ ist, dann können wir einfache analytische Methoden verwenden um ein globales Minimum zu finden. Wenn aber Θ eine Matrix oder dergleichen ist, und $L(D(x), M(\theta)(x))$ keine lineare Funktion, dann muss man kompliziertere Formen der Optimierung verwenden; für uns meistens das sog. Gradientenverfahren.

(5) ist also nicht sonderlich nützlich was das praktische Verfahren angeht. Aber es zeigt uns sehr schön, worum es beim ML eigentlich geht: nämlich einfach Lösung für (5) zu finden; und wenn das nicht geht, dann eine passable

Approximierung hierzu. In den Fällen, die hier interessant sind, ist θ übrigens nie einfach eine reelle Zahl, sondern wesentlich komplexer.

Unsere Annahme ist hierbei die folgende: wir glauben, dass wenn ein Modell die Daten gut beschreibt ($\cong$ geringe Kosten), dann wird es auch bei neuen Beobachtungen (die also nicht in unseren Daten vorkommen) das richtige tun. Diese Annahme kann aber aus verschiedenen Gründen falsch sein; wir werden das in den folgenden Abschnitten besprechen.

2.2 Meta-Parameter, Overfitting, Underfitting

Bei linearer Regression geht es darum, Parameter zu schätzen, wie immer beim maschinellen Lernen. Es gibt aber auch *Meta-Parameter*, die wir *apriori* festlegen, und die durch die Daten nicht beeinflusst werden. In diesem Fall ist das der Grad der Funktion, die wir für die Regression nutzen, also linear, quadratisch etc. Um hier den richtigen Wert zu finden, müssen wir die Daten richtign einschätzen: ist der Grad zu niedrig, finden wir nicht die passende Generalisierung (underfitting); ist der Grad zu hoch, übergeneralisieren wir. Um Meta-Parameter richtig zu schätzen – bzw. um Anhaltspunkte zu haben, wie sie am besten liegen in einer gewissen Klasse von Problemen – nimmt man üblicherweise eine Trennung vor von Trainingsdaten und Testdaten. Die Testdaten werden nicht genutzt, um das System zu optimieren, aber hinterher wird das System auf den Testdaten *evaluiert*. Auf diese Weise kann man prüfen (bis zu einem gewissen Maß, ob wir die korrekten Generalisierungen getroffen haben.

Es gibt für das ML zwei große Probleme, oder besser gesagt: zwei Arten, wie unsere Ergebnisse danebenliegen können, die eine ist *overfitting*, die andere *underfitting*. Einfach gesagt handelt es sich um folgende Probleme: *overfitting* bedeutet, dass wir keine ausreichenden Generalisierungen treffen, weil wir zu sehr an den Daten bleiben; *underfitting* bedeutet, dass die Dateen nicht genügend berücksichtigen und daher die Muster nicht erfassen. *Overfitting* geschieht, wenn unsere Modellklasse zu mächtig ist, also zu viele Parameter hat. *Underfitting* bekommen wir, wenn unsere Modellklasse nicht mächtig genug ist, also die Variation der Daten gar nicht erfassen kann.

Man kann das gut anhand von einem Beispiel erklären: nehmen wir an, wir sehen eine Elster im Park ein Nest bauen. Daraus kann man verschiedene Generalisierungen ziehen:

(1) a. Elstern bauen im Park Nester.
b. Elstern bauen Nester.
c. Vögel bauen Nester.

In diesem Fall wäre a. eine Form von Overfitting: wir haben einen Parameter zuviel, nämlich den Ort, der keine Rolle spielt, und daher entgeht uns die richtige Generalisierung b. Umgekehrt ist c. eine falsche Generalisierung, denn wir haben einen Parameter zuwenig, die Vogelart, die wir brauchen um eine korrekte Generalisierung zu treffen. Es geht also darum, die richtige Zahl von Parametern zu finden (in unserem Beispiel: Faktoren die relevant

sind), um die richtigen Generalisierungen zu finden. Dafür gibt es allerdings kein Patentrezept, wie wir auch im obigen Beispiel sehen: oft hilft es einfach nur, wenn wir domänenspezifisches Wissen haben, dass wir anwenden (mehr dazu gleich).

Ein weiteres Problem hierbei ist, dass es oft Störparameter gibt, also Parameter, die nicht relevant sind für das was wir suchen. das können typischerweise Meßfehler sein, aber auch durchaus systematische Dinge: nehmen wir an, wir suchen den Erwartungswert einer Normalverteilung. Dann ist die Varianz dieser Verteilung ein Störparameter, da sie durchaus unsere Beobachtungen beeinflusst, aber keine Relevanz hat für das was wir suchen. Da es oft diesen Störparameter gibt, ist es auch nicht immer wichtig, dass unsere Modelle die Daten exakt reproduzieren: das wäre nämlich oft bereits eine Form von overfitting. Viel wichtiger ist, dass die Abweichung nicht systematisch ist, und das wir eben nicht zuviele Parameter haben.

2.3 Parameter und Hyperparameter, Test und Trainingsdaten

Die Frage, ob wir over- oder underfitting haben, läßt sich nicht beantworten, indem wir nur einen Datensatz betrachten. Stattdessen teilen wir den Datensatz in zwei Teile: nämlich die **Trainingsdaten** und die **Testdaten**. Wir nutzen die Trainingsdaten für die Optimierung des Modells (das eigentliche Lernen). Wenn das abgeschlossen ist, dann schauen wir, wie gut das Modell auf den Testdaten funktioniert. Die Idee dahinter ist folgende: wenn wir unser Modell auf den Trainingsdaten overfitten, dann wird es auf den Testdaten schlechter abschneiden; dasselbe gilt fürs underfitten. Wichtig ist hierbei: das Modell darf während des Trainings die Testdaten nicht sehen, und wir müssen streng verhindern, dass Informationen aus den Testdaten im Training verwendet werden. Wenn wir nun zuviele Parameter verwenden, dann bekommen wir gute Ergebnisse auf den Trainingsdaten, aber nicht auf den Testdaten, denn diese überflüssigen Parameter konnten ja nicht in Hinblick auf diese Daten optimiert werden. Gleichzeitig gibt es natürlich keinen Grund, warum wir underfitten sollten, denn wenn ein Parameter relevant ist, wird er sowohl für Trainings- als Testdaten relevant sein. Diese Methode ist also sehr geeignet um das oben beschriebene Problem zu umgehen.

Die Partition der Daten in Training und Test sollte rein zufällig geschehen; üblicherweise nimmt man ein Verhältnis 4:1 an. Es kann evtl. Probleme geben, wenn wir nicht genügend haben, wir also keine Daten für den Test "entbehren" können. In diesem Fall nutzt man die Methode der **cross-validation**: wir partionieren unsere Daten in Test- und Trainingssatz, wobei der erstere viel zu klein ausfällt. Das wird dann allerdings immer wieder iteriert (z.B. ist jeder Datenpunkt einmal Trainingsatz). Das liefert jedesmal ein (etwas) anderes Modell, aber am Ende können wir über die Ergebnisse mitteln und so sehen, ob das Modell insgesamt stimmt.

Das bringt uns auf eine weitere wichtige Unterscheidung: bei cross-validation haben wir streng genommen jedesmal andere Parameter; was gleich bleibt sind die **Hyperparameter**. Hyperparameter sind Eigenschaften des Modells, die nicht von den Daten abhängen, z.B. dass das Modell eine lineare Funktion ist (ein Polynom). Übrigens ist auch die Anzahl und Art der Parameter, die unser Modell hat, ein Hyperparameter. Mit cross-validation zeigen wir also, dass wir korrekte Hyperparameter gewählt haben.

Hier gibt es allerdings eine Kleinigkeit zu beachten: nehmen wir an, es gibt einen Datensatz D, mit dem wir wiederholt verschiedene Modelle ($\cong$

Hyperparameter) testen, indem wir ihn auf verschiedene Art und Weise in Test- und Trainingssatz spalten. Auf diese Weise schließen wir overfitting für die Parameter aus. Eine andere Sache, die jedoch passieren kann, ist dass wir auf diese Art und Weise overfitting für die Hyperparameter bekommen; denn diese werden immer wieder auf demselben Datensatz trainiert. Das ist insbesondere ein Problem für Gebiete, auf denen Daten nur schwer zu bekommen sind. Deswegen ist es für alle Gebiete wichtig, dass wir beständig neue Daten haben.

3 Vorspiel 1: Lineare Modelle

3.1 Die einfache lineare Regression

Ein sehr wichtiges und einfaches Verfahren des maschinellen Lernens ist die lineare Regression. Hier wird versucht, eine Funktion mittels einer linearen Funktion zu approximieren. Etwas allgemeiner verwendet man lineare Regression für die Approximation mittels Polynomialfunktionen beliebigen Grades. Nehmen wir erstmal eine einfache lineare Funktion $f : \mathbb{R} \to \mathbb{R}$, die die Form haben soll:

$$f(x) = ax + b \tag{7}$$

wobei $a, b \in \mathbb{R}$ die Parameter sind. In diesem Fall muss unser Datensatz $D \subseteq \mathbb{R} \times \mathbb{R}$ einfach eine Abhängigkeit zweier reellwertiger Parameter darstellen. Wir setzen als Konvention

$$D = \{(a_1, b_1), ..., (a_n, b_n)\}$$

und für $x = a_i$ schreiben wir y_x für b_i, also der Wert den D x zuweist. Wir suchen nun diejenige lineare Funktion, die die Differenz minimiert, also

$$argmin_{a,b\in\mathbb{R}} \sum_{(x,y)\in D} (ax + b - y_x)^2 \tag{8}$$

Das Quadrat ist dafür da, dass Werte positiv werden – sonst würden sich negative und positive Abweichungen ausgleichen. Damit gewichten wir natürlich weitere Abweichungen stärker, was nicht unbedingt erwünscht ist; allerdings gibt es kaum andere Möglichkeiten: die Betragsfunktion $|-|$ ist nicht differenzierbar, wir brauchen allerdings die erste Ableitung der Funktion, wie wir unten sehen werden.

Wir machen nun einen Trick: eigentlich sind die Parameter a, b festgelegt, während x das variable Argument der Funktion ist. Weil wir aber nur an denjenigen x interessiert sind, die in unserem Datensatz auftauchen (d.h. endlich viele), während wir alle reellen Parameter berücksichten müssen. Daher ist die Funktion, die wir minimieren müssen, eigentlich folgende:

$$\sum_{(a,b)\in D} (ax + y - b)^2 = (a_1x + y - b_1)^2 + ... + (a_nx + y - b_n)^2 \tag{9}$$

Hier haben wir einfach die Konstanten und Variablen vertauscht, und daraufhin eine arithmetische Umformung vorgenommen. Am Ende bekommen wir die einfache Form (denn $a_1, ..., a_n, b_1, ...b_n$ sind einfache gegebene Konstanten)

$$f(x,y) = ax^2 + by^2 + cxy + dx + ey + f \tag{10}$$

Denn alle Summanden haben eine dieser Variablenformen als Koeffizient, und wir suchen einfach

$$argmin_{x,y\in\mathbb{R}} ax^2 + by^2 + cxy + dx + ey + f \tag{11}$$

Das berechnet man mit der gewohnten Methode: wir bilden (in diesem Fall partielle) Ableitungen und konstruieren damit den Gradienten. Das ist natürlich besonders einfach:

$$\nabla f(x,y) = ((2ax + cy + d), (2by + cx + e)) \tag{12}$$

Wir haben also 2 Gleichungen, die wir auf 0 setzen müssen:

$$2ax + cy + d = 0 \tag{13}$$

$$2by + cx + e = 0 \tag{14}$$

Wir haben 2 Gleichungen und 2 Variablen, also eine Lösung:

$$x = -\frac{cy + d}{2a} \tag{15}$$

also

$$2by - \frac{c^2y + cd}{2a} + e = 0 \leftrightarrow \tag{16}$$

$$y = \frac{cd - 2ae}{4ab - c^2} \tag{17}$$

Wir haben also die Nullstelle für x, y berechnet, und wissen somit, wie wir die Funktion minimieren können.

3.2 Der komplexe lineare Fall

Im komplexeren lineare Fall nehmen wir an, dass

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0 \quad (18)$$

f ist also nun ein Polynom, keine lineare Funktion mehr. Warum ist das immer noch lineare Regression? Weil wir, bei der eigentlichen Regression, also der Suche nach den Parametern die die beste Funktion ausmachen, x als eine Konstante behandeln (wir setzen nämlich Datenpunkte ein, bekommen also einfach konstante Werte in $\mathbb{R}$), während die eigentlichen Variablen die Werte $a_0, ..., a_n$ sind. In diesen Werten ist die resultierende Funktion nach wie vor linear – wir haben also einen Fall der etwas komplexer ist als der vorhergehende, aber nach wie vor durch die Lösung linearer Gleichungssysteme lösbar ist.

$$f(x_0, ..., x_n) = \sum_{(a,b) \in D} (a^n x_n + a^{n-1} x_{n-1} + ... + x_0 - b)^2 \quad (19)$$

Hier sind a^n etc. und b fixe reelle Zahlen, während die Variablen nur linear auftreten. Wir müssen nun

$$\nabla f : \mathbb{R}^{n+1} \to \mathbb{R}^{n+1} \quad (20)$$

konstruieren, kriegen also ein lineares Gleichungssystem mit $n+1$ Variablen und $n+1$ Gleichungen, das wir entsprechend lösen können. Lineare Regression lässt sich also mit elementaren mathematischen Methoden lösen, und das ist der große Vorteil dabei.

Eine weitere Erweiterung, die ohne große Probleme funktioniert, ist folgende: anstatt einer Funktion $f : \mathbb{R} \to \mathbb{R}$ suchen wir eine Funktion $f : \mathbb{R}^n \to \mathbb{R}$, die die Form hat

$$f(x_1, ..., x_n) = a_1 x_1 + a_2 x_2 + ... + a_n x_n + b \quad (21)$$

Auch das geht ohne große Probleme mit den obigen elementaren Methoden, und auch die Erweiterung auf Polynomialfunktionen (auch wenn natürlich alles etwas schwieriger wird).

3.3 Lineare Regression in R

In R gibt es ein einfaches Kommando zur (einfachen) lineaeren Regression, nämlich `lm`. Erstmal brauchen wir Daten; dazu nehmen wir eine Menge $D \subseteq \mathbb{R}^2$. In R geht das einfacher, wenn man zwei Vektoren nimmt:

$$v_1 = (x_1, ..., x_n),\ v_2 = (y_1, ..., y_n), \text{ wobei } D = \{(x_1, y_1), ..., (x_n, y_n)\}$$

Denn hiermit kann man eine einfache Funktion nutzen:

> x <- c(-4,-2.8,-1.5,0,0.7,1.9,2.3)
> y <- c(7,4.3,5.2,3.8,3.6,2,0.8)
> plot(x,y,xlim = c(-5,5),ylim=c(-7,7))

Hier werden also die Werte von `x` gegen `y` geplottet, wobei die Zuordnung anhand der Stelle im Vektor erfolgt. Nun möchten wir diese Funktion linear approximieren, also mittels eines Modelles

(22) $f(x) = ax + b$

Das geht in R ganz einfach mit dem Befehl:

> lm1 <- lm(y ∼ x)

Wir haben hier das Modell lm1, und das soll y als linear abhängige Variable von x vorhersagen.

> lm1
Call:
lm(formula = y x)

Coefficients:
(Intercept) x
3.4308 -0.7896

Wir bekommen also den *intercept*, eine konstante die uns sagt wo die Funktion die y-Achse kreuzt, und den *slope*, also die Steigung, der die lineare Anhängigkeit von x liefert; wir haben also:

(23) $lm1(x) = -0.7896x + 3.4308$

Das ist also die beste lineare Approximation unserer Daten. Wie gut sie ist kann man mit der Funktion `residuals` sehen, die jeweils liefert:

(24) $residuals(f) = (f(x_1) - y_1, ..., f(x_n) - y_n)$

In unserem Fall also:

> residuals(lm1)
1 ...
0.41079392 ...

Man kann die Regressionsfunktion auch schön visualisieren, mittels:

> abline(lm1, col = "red")

Um Modelle höherer Ordnung in R zu bekommen, gibt es meines Wissens nach keinen direkten Befehl. Man kann das aber recht einfach machen. Zunächst muss man wissen, wie man eine Variable x in Abhängigkeit mehrerer Variablen setzt. Das geht ganz einfach:

> lm2 <- lm(y ~ x+z)

Jetzt müssen wir nur noch den passenden Vektor z erstellen:

> z <- 1:7
> for(i in 1:7)z[i]=(x[i])^2
> lm2 <- lm(y+ ~ x + z)

In diesem Fall bekommen wir eine Polynom zweiter Ordnung für unsere Funktion; wir können die Ordnung weiter erhöhen, bis unsere Residuen bei 0 liegen werden. Die große Frage ist: wird unser Modell dadurch besser?

3.4 Beschränkungen linearer Modelle, lineare Separierbarkeit

Lineare Modelle sind sehr intuitiv und leicht zu handhaben (trainieren, optimieren). Sie haben aber auch wichtige Beschränkungen – so wie sich immer Vorteile und Nachteile auswiegen. Eine sehr alte und bekannte Feststellung ist, dass gewisse sehr einfache Boolesche Funktionen nicht mittels linearer Modelle berechnet werden können. Das bekannteste Beispiel hierfür ist das exclusive logische oder, oft mit XOR bezeichnet. XOR ist eine binäre Funktion

$$\{0,1\} \times \{0,1\} \to \{0,1\}$$

definiert durch folgende Tabelle:

XOR	0	1
0	0	1
1	1	0

Diese Funktion ist besonders durch eine Eigenschaft: von

$$\text{XOR}(0,0)$$

ausgehend vergößert sich das Ergebnis mit einer Vergößerung in beiden Argumenten einzeln (also monoton steigen). Aber ausgehend von

$$\text{XOR}(0,1) \text{ und } \text{XOR}(1,0)$$

ist es in den beiden 0-Komponononenten *monoton fallend*. Anders gesagt: ob XOR in der ersten/zweiten Komponononente monoton steigend/fallend ist, hängt von der jeweils anderen Komponente ab!

Um zu sehen warum das nicht gehen kann, brauchen wir den Begriff der **linearen Separierbarkeit**. Nimm eine lineare Funktion

$$f : \mathbb{R}^n \to \mathbb{R}.$$

Dann ist die Menge

$$\{x : f(x) \geq \alpha\} \text{ für ein beliebiges } \alpha$$

trennbar durch eine *Hyperebene im Vektorraum* $\mathbb{R}^n$ (also im Fall $n = 1$ ein Punkt, im Fall $n = 2$ eine gerade, im Fall $n = 3$ eine Ebene etc.).

Wenn man die XOR-Funktion in eine Ebene zeichnet (bzw. die beiden Datenpunkte), dann ist klar dass es keine Linie geben kann, die die Werte ≥ 1 von denen < 1 separiert.

Das ist eine wichtige Beobachtung, die die Beschränkungen linearer Funktionen zeigt. Falls wir aber lineare Funktionen im allgemeineren Sinn nutzen (also Polynome), stimmt die lineare Separierbarkeit bereits nicht mehr; allerdings bleibt die Separierbarkeit durch eine – je nach Grad des Polynoms – konstant begrenzte Menge von Hyperebenen, und das Problem des XOR lässt sich neu stellen. Vergleiche auch: lineare Regression vs. nearest neighbour regression.

Anhand dieses Beispiel kann man auch die Wichtigkeit nichtlinearer Funktionen zeigen: wenn wir unsere Eingabedaten **nicht-linear transformieren**, z.B. mittels der Funktion

$$\phi(x, y) = (x \cdot y, x + y) \tag{25}$$

dann bekommen wir eine Funktion

XOR	0	1	2
0	0	1	-
1	-	-	0

Das Problem hierbei ist: die passende nichtlineare Transformation zu finden setzt im Normalfall voraus, dass wir das Problem um das es sich handelt, bereits gut verstehen, und das wir die gesuchte Generalisierungen bereits in groben Umrissen kennen. Genau das möchten wir beim maschinellen Lernen aber nicht voraussetzen!

3.5 Abschluss unter Komposition

Eine wichtige Eigenschaft linearer Funktionen ist ihr Abschluss unter **Komposition**, dann heißt: wenn wir ein lineares Modell auf ein lineares Modell applizieren, dann resultiert daraus eine lineare Funktion. Wir definieren, wie üblich, $f \circ g$ durch $f \circ g(x) = f(g(x))$. Der Abschluss unter Komposition sagt nun: falls f, g lineare Funktionen sind, dann ist auch $f \circ g$ eine lineare Funktion. Das kann man für den einfachen Fall sehr schön zeigen: sei

$$f(x) = ax + b \tag{26}$$

$$g(x) = cx + d \tag{27}$$

Nun ist

$$\begin{aligned} f \circ g(x) &= a(cx + d) + b \\ &= acx + ad + b \\ = (ac)x + adb \end{aligned}$$

also offensichtlich eine lineare Funktion. Man kann das, nach demselben Muster, auch für komplexere Funktionen zeigen (Polynome, Matrizen auf Vektoren). Die Schlussfolgerung hieraus ist: wir werden nicht mächtiger, wenn wir lineare Modelle hintereinander schalten. Das bedeutet auch: die nichtlinearen Aktivierungsfunktionen in neuronalen Netzen sind entscheiden – wenn die nicht wären, dann wäre das ganze Netz ein lineares Modell. Die Mächtigkeit von neuronalen Netzen basiert also auf der Abfolge

lineares Modell - nichtlineares Modell - ... - lineares Modell - nichtlineares Modell

Wenn wir die Kombination lineares Modell+nichtlineares Funktion nehmen, dann haben wir keinen Abschluss unter Komposition: es gibt also eine echte Hierarchie von Klassen von Funktionen

$$l_1(x) \tag{28}$$

$$n_1 \circ l_1(x) \tag{29}$$

$$l_2 \circ n_1 \circ l_1(x) \tag{30}$$

$$n_2 \circ l_2 \circ n_1 \circ l_1(x) \tag{31}$$

$$... \tag{32}$$

wobei l_i jeweils eine lineare Funktion ist, n_i jeweils eine nichtlineare Funktion. Je mehr solche Funktionen wir hintereinander schalten, desto mächtiger wird die Klasse der Funktionen, die wir auf diese Weise berechnen können. Hierauf basiert die Mächtigkeit neuronaler Netze und das universelle Approximationstheorem.

3.6 Die Logistische Regression und Aktivierungsfunktionen

Logistische Regression ist im engeren Sinne keine Regression, sondern Klassifikation; es ist aber eine Klassifikation, die auf linearer Regression aufbaut. Logistische Regression funktioniert im einfachen Fall, wenn wir 2 Klassen haben, zwischen denen wir auswählen, z.B. A und B. Unser Problem ist also folgendes: wir möchten eine Eingabe in $\mathbb{R}$ nach A oder B klassifizieren. Wir machen das mittels der logistischen Sigmoid-Funktion

$$S(x) = \frac{1}{1+e^{-x}} \tag{33}$$

Einige Beobachtungen: wir haben

- $lim_{x\to-\infty}S(x) = 0$
- $lim_{x\to\infty}S(x) = 1$
- $S(x) \in (0,1)$ f.a. $x \in \mathbb{R}$
- $S(x)$ ist streng monoton steigend in x

```
> f <- function(x){1/(1+exp(-x))}
> plot(f,xlim=c(-10,10),ylim=(0,1))
```

Allgemeiner nennt man solche Funktionen **Sigmoidfunktionen**. Ein anderes Beispiel für eine solche Funktion ist

$$T(x) = \frac{x}{1+|x|} \tag{34}$$

$$U(x) = \frac{x}{1+x^2} \tag{35}$$

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 - \frac{2}{e^{2x}+1} \tag{36}$$

Was allen diesen Funktionen gemeinsam ist, ist dass sie im Bereich um 0 relativ schnell von 0 bzw. -1 Richtung 1 wechseln, sonst für groe positive/negative Zahlen langsam Richtung 1/0 konvergieren. Es gibt also nur einen kleinen Bereich, in dem eine nicht extreme Änderung im Argument eine

signifikante Änderung im Wert verursacht; ansonsten gibt es relativ schnell eine Sättigung. solche Prozesse findet man oft in der Natur, z.B. bei Populationen. Solche logistischen Funktionen haben allerdings die Einschränkung dass sie nur $\mathbb{R} \to \mathbb{R}$ definiert sind. Aktivierungsfunktionen spielen in neuronalen Netzen eine große Rolle, und werden normalerweise auf die einzelnen Komponenten von Vektoren angewendet.

Wenn wir nun 2 Klassen haben, dann können wir einfach sagen: wir interpretieren den Wert als eine **Wahrscheinlichkeit**, also:

$$P(X = A|x) = S(f(x)) \tag{37}$$

wobei f eine lineare Funktion ist, die wir mittels linearer Regression induzieren. Wir transformieren also eine Funktion in die reellen Zahlen zu einer Funktion nach $[0, 1]$:

$$f : \mathbb{R} \to \mathbb{R} \Longrightarrow S \circ f : \mathbb{R} \to [0, 1]$$

Das Ergebnis können wir dann als Wahrscheinlichkeit interpretieren. Das gibt uns nun einen Klassifikator:

$$C(x) = \begin{cases} A, \text{ falls } P(X = A|x) > 0.5 \\ B \text{ andernfalls} \end{cases} \tag{38}$$

Das funktioniert natürlich nur, falls wir nur zwei Klassen (hier A, B) zur Verfügung haben. Intuitiv bilden wir hier die Eingabe auf eine Wahrscheinlichkeit ab, und Klassifizieren nach Maximum Likelihood. Natürlich kann auch das sehr leicht generalisiert werden auf beliebige Eingaben $\mathbf{x} \in \mathbb{R}^n$.

3.7 Bedeutung

Die Bedeutung der logistischen Regression ist erstmal folgende: wir suchen eine Klassifikationsfunktion

$$f : \mathbb{R}^n \to \{0, 1\}$$

Wir tun das aber mittels zweier Zwischenschritte:

$$\mathbb{R}^n \underset{f}{\Longrightarrow} \mathbb{R} \underset{S}{\Longrightarrow} \{0, 1\}$$

Die zugrundeliegende Annahme ist hierbei, dass es einen kritischen Bereich gibt, auf dem die Klassifikation von 0 zu 1 springt und umgekehrt ist; aber damit man das sieht, muss man zunächst eine **numerische Transformation** durchführen, nämlich die lineare Regression. Genauer gesagt, die lineare Funktion f kann die numerischen Werte so transformieren, dass gilt:

$$\text{Falls } f(x) > s, \text{ dann } C(x) = 1, \text{ und falls } f(x) \leq s, \text{ dann } C(x) = 0$$

Das bedeutet, wir haben eine *quasi-lineare Abhängigkeit* von unseren Eingabedaten und den Klassen. Das kann man noch genauer fassen, es gibt nämlich eine genaue Beschreibung wenn wir den **Hyperparameter** des **Grades der Funktion** mitberücksichtigen. Nehmen wir an, wir suchen eine Funktion $f : \mathbb{R} \to \{0, 1\}$. Wir wissen, dass

- Eine lineare Funktion ist hat eine Gerade als Graphen, kann also einen gewissen Wert nur einmal annehmen
- Ein Polynom zweiten Grades hat einen Wendepunkt, kann einen gewissen Wert höchstens zweimal annehmen.
- ...
- Ein Polynom $n$ten Grades hat n Wendepunkte, kann einen gewissen Wert höchstens n-mal annehmen.

Dann heißt das soviel wie:

- Falls wir eine lineare Funktion haben, dann gibt ein $\alpha \in \mathbb{R}$, so dass falls $x < \alpha$, dann $f(x) = 1$ (bzw. 0), ansonsten $f(x) = 0$ (bzw. 1). Wir spalten also den Eingaberaum $\mathbb{R}$ in zwei Teile.

- Falls wir ein Polynom zweiten Grades haben, dann gibt $\alpha_1, \alpha_2 \in \mathbb{R}$, so dass falls $\alpha_1 < x\alpha_2$, dann $f(x) = 1$ (bzw. 0), ansonsten $f(x) = 0$ (bzw. 1). Wir spalten also den Eingaberaum $\mathbb{R}$ in drei Teile.
- ...
- Falls wir ein Polynom n-ten Grades haben, haben wir $\alpha_1, ..., \alpha_n$, die $\mathbb{R}$ in $n + 1$ Teile spalten, und wir klassifizieren auf dieser Basis.

3.8 Lernen & Anwendung

Wir betrachten folgendes Beispiel: wir möchten, als abhängige, diskrete Variable vorhersagen, ob ein Student die Prüfung besteht (0 oder 1; abhängig bedeutet nur: wir möchten das vorhersagen). Als Prädiktor nutzen wir die Anzahl der Stunden, die der Student gelernt hat. Unsere Regressionsfunktion soll eine einfache lineare Funktion sein. Unser Datensatz sieht nun wie folgt aus (nennen wir den Datensatz $D1$):

Stunden gelernt	4	5	6	7	8	9	10
Bestanden	0	0	1	0	1	1	1

Eine lineare Regression liefert hier folgendes Ergebnis (danke R):

(39) $f(x) = 0.1786x - 0.6786$

Wir haben also eine positive Abhängigkeit von Lern-Stunden und Klausur-Bestehen (zum Glück); aber dieses Ergebnis ist noch unbefriedigend: wir können das nicht sinnvoll als Wahrscheinlichkeit interpretieren. Wir setzen aber nun diesen Term ein in unsere Aktivierungsfunktion, und defineren:

(40) $$P(\text{bestehen}|n \text{ Stunden lernen}) = \frac{1}{1 + e^{-f(x)}} = \frac{1}{1 + e^{-(0.1786x - 0.6786)}}$$

Das liefert uns nun ordentlich Werte:

Stunden	4	5	6	7	8	9	10
P(bestehen)	0.508	0.553	0.597	0.639	0.679	0.717	0.752

Das ist schon besser, aber nicht wirklich optimal, insbesondere stört die Asymmetrie (bei 4 haben wir bereits eine ziemlich hohe Wahrscheinlichkeit!).

Die logistische Funktion interagiert mit der linearen auf eine Art und Weise, die nicht optimal ist. Wir sollten also stattdessen folgendes suchen:

$$(41) \quad argmin_{a,b\in\mathbb{R}} \sum_{(x,y)\in D} \frac{1}{1+e^{-(ax+b)}} - y$$

Wir nutzen also die Aktivierungsfunktion, um den Unterschied zwischen 0 und 1 sichtbar zu machen, da eine lineare Funktion hierzu nicht in der Lage ist. Um 41 auszurechnen gibt reichen keine elementaren Methoden, man muss also etwas komplexere numerische Optimierung anwenden (der Computer kann das). R macht das mit dem Kommando:

> glm(x ∼ y, family = binomial())

Nun definieren wir:

> $g <-$ function(x){1/(1+exp(-(1.251*x-8.113)))}

Das sollte nun stimmen; wir bekommen dementsprechend:

Stunden	4	5	6	7	8	9	10
P(bestehen)	0.0427	0.135	0.353	0.656	0.869	0.959	0.988

Tadaa! Wir bekommen also eine Wahrscheinlichkeit. Was aber interessanter ist, ist folgende Frage: gegeben diese Ergebnisse, wie ist die Wahrscheinlichkeit, dass die Abhängigkeit des Bestehens von der Lernzeit reiner Zufall ist? Das ist eine klassische statistische Frage, und im Zusammenhang mit logistischer Regression gibt es den sog. **Wald-test**:

$$(42) \quad \frac{(\theta_{ML}-\theta)^2}{var(\theta_{ML})}$$

Er gibt die Wahrscheinlichkeit, dass die gegebene Verteilung zufällig ist, also dass es keine Abhängigkeit der beiden Variablen (Lernzeit / Bestehen) gibt. Das ist verwandt mit dem Test des Likelihood-Verhältnisses, was ein wenig einfacher zu verstehen ist. Seien

- ω unsere Beobachtungen, also die Daten in D1;
- H_0 die Nullhypothese, also Unabhängigkeit der Variablen: Lernzeit hat keinen Einfluss auf das Bestehen, Wahrscheinlichkeit von Bestehen oder Nicht-bestehen wird durch Maximum Likelihood geschätzt.

- H_1 die Alternativhypothese, also unsere durch logistische Regression berechnete bedingte Verteilung.

Dann haben wir den Likelihood-Quotienten:

$$R(\omega) := \frac{P(\omega|H_0)}{P(\omega|H_1)} \qquad \text{(Sonderfall für } P(\omega|H_1) = 0) \tag{43}$$

Hierauf kann man nun den Schwellentest S_t anwenden, $t > 0$, und üblicherweise $t = 0.05$. Zur Erinnerung: das ist der Test, der sich für H_0 entscheidet falls $R(\omega) > t$, und für H_1 andernfalls, also:

$$S_t(\omega) = \begin{cases} H_0, \text{ falls } R(\omega) > t \\ H_1 \text{ andernfalls.} \end{cases} \tag{44}$$

Das können wir/Sie nun ausrechnen:

3.9 Ein Vergleich: nearest neighbour Regression

Nearest neighbour regression ist eine denkbar simple Methode: wir haben eine Funktion $f : V \to C$, wobei V ein Vektorraum ist, und C eine (endliche) Menge von Klassen; wir haben eine Menge $D \subseteq V \times C$ von Datenpunkten, mittels derer wir die Funktion f "lernen" sollen. Was wir hierfür erstmal brauchen ist der Begriff der **Norm**:

Eine Norm, definiert auf einem Vektorraum V ist das eine Funktion

$$\| - \| : V \to \mathbb{R}_0^+$$

es werden also beliebige Vektoren auf einen nicht-negativen Wert abgebildet. Zusätzlich muss $\| - \|$ noch folgende Bedingungen erfüllen f.a. $\vec{v} \in V$, $\lambda \in \mathbb{R}$.

1. $\|\vec{v}\| = 0 \Rightarrow \vec{v} = 0_V$ (die 0 des Vektorraums, neutral für Addition)
2. $\|\lambda \cdot \vec{v}\| = |\lambda| \cdot \|\vec{v}\|$, wobei $|\lambda|$ der Betrag ist (respektiert Skalarmultipikation)
3. $\|\vec{v} + \vec{w}\| \leq \|\vec{v}\| + \|\vec{w}\|$ (allgemeine Dreiecksungleichung)

Die intuitivste Norm ist die *euklidische*, die jedem Vektor seine **Länge** zuweist (wenn wir einen Vektor als eine Linie vom Ursprung auf seine Koordinaten (im n-dimensionalen Raum) auffassen. Diese Norm basiert auf einer Verallgemeinerung des Satz des Pythagoras:

$$\|(v_1, ..., v_n)\| = \sqrt{v_1^2 + ... + v_n^2} \tag{45}$$

In dieser geometrischen Interpretation wird Bedingung 3 zur **Dreiecksungleichung**: in jedem rechwinkligen Dreieck ist die Länge der Hypothenuse geringer als die Summe der Länge der Katheten. Es gibt aber noch viele weitere Normen, z.B. die sog. p-Norm, wobei $p \geq 1$ eine reelle Zahl ist:

$$\|(v_1, ..., v_n)\|_p = \left(\sum_{i=1}^{n} |v_i|^p \right)^{\frac{1}{p}} \tag{46}$$

Für $p = 1$ vereinfacht sich das zu

$$\|(v_1, ..., v_n)\|_1 = \sum_{i=1}^{n} |v_i| \tag{47}$$

Das ist die sog. Manhattan-Norm, weil man immer rechtwinklig um die Blocks fahren muss – das gibt im 2-dimensionalen Fall also die kürzeste Strecke in Manhattan an.

Darauf basiert der Begriff der Metrik; jede Norm induziert eine Metrik d mittels

(48) $d(\vec{v}, \vec{w}) = \|\vec{v} - \vec{w}\|$

wobei natürlich

(49) $\vec{v} - \vec{w} = (v_1 - w_1, ..., v_i - w_i)$

Es ist nicht schwer zu sehen dass gilt:

- $d(x, y) \geq 0$ (positiv)
- $d(x, y) = d(y, x)$ (symmetrisch)
- $d(x, y) \leq d(x, z) + d(z, y)$ (Dreiecksungleichung)

Die **euklidische Distanz** ist definiert durch die euklidische Norm, mit

(50) $d_2(\vec{v}, \vec{w}) = \|\vec{v} - \vec{w}\|_2$

Nun machen wir einfach folgendes: gegeben einen beliebigen Vektor $\vec{v}$, eine Norm $\| - \|$ mit Metrik d, definieren wir

(51) $nn_D(\vec{v}) = argmin_{(\vec{w},c)\in D} d(\vec{v}, \vec{w})$

Wir suchen uns also den **nächsten Nachbarn** nach unserer Metrik. Als nächstes machen wir das denkbar naheliegendste: wir machen genau das, was der nächste Nachbar macht (Einfachheit halber fassen wir unseren Datensatz D als partielle Funktion $D : V \to C$ auf)

(52) $NNR_D(\vec{v}) = D(nn_D(\vec{v}))$

In Wort: $NNR_D : V \to C$ ist die nearest neighbour regression über D, und diese (vollständige) Funktion funktioniert, indem sie für jeden Eingabevektor $\vec{w}$ zunächst den (nach Metrik d nächstliegenden Vektor $\vec{v}$ im Datensatz findet (ein endliches Suchproblem), um ihm dann dieselbe Klasse zuzuordnen, die auch $\vec{v}$ zugeordnet wird.

Ein einfaches Beispiel wäre z.B. eine Sprach- oder Bilderkennung: bei Bildern wäre das Pixelbild ein Vektor, in dem jeder Pixel eine Komponente ist, und der Wert ist die Farbe der Pixel. Bei Spracherkennung kann man die verschiedenen Frequenzbereiche F1-F3 auf einen Vektor verteilen, mit dem Wert als Frequenz; die Klassifikation wäre dann die Frage: welcher Gegenstand ist auf dem Bild abgebildet (aus einer endlichen Menge), bzw. welcher laut wird geformt? NNR ist einfach folgende Methode: finde den ähnlichsten Punkt in unserem Datensatz und klassifiziere entsprechend.

Was ganz interessant ist: NNR kann nicht durch eine lineare Funktion beliebiger Ordnung modelliert werden. Das sieht man sehr leicht an einem Beispiel: man nehme einen Datensatz

$$D = \{((0,0),a),((0,1),b),((0,2),a),((0,3),b),...\}$$

NNR wird hier beliebig oft wechseln können zwischen a und b; jedes lineare Modell, wie z.B. bei logistischer Regression, wird nur eine konstante Anzahl von wechseln ermöglichen können. Allerdings gilt das nur *apriori* und ohne Trainingsdaten: denn wenn wir eine NNR haben zusammen mit einem Datensatz D, dann wird es auch nur eine konstante Zahl an wechseln geben, einfach weil die Datenmenge endlich ist!

Wir sehen hier aber, was ein Modell des maschinellen Lernens ausmacht: es ist eine Funktion von (Trainings)Daten zu einer (Klassifikations- oder Regressions-)Funktion.

$$\text{(Abstraktes) ML-Modell} \underset{Daten}{\Longrightarrow} \text{(Konkrete) Funktion} \underset{Eingabe}{\Longrightarrow} \text{Ausgabe}$$

Übrigens gilt auch bei dieser Methode: man sollte die Daten aufspalten in Trainings- und Testdaten, um festzustellen ob die Methodik angemessen ist. Das kann in manchen Fällen der Fall sein, in manchen nicht.

Stunden gelernt	4	5	6	7	8	9	10
Bestanden	0	0	1	0	1	1	1

Wenn wir wiederum diese Tabelle betrachten, dann sehen wir wir $NNR(5.6) = 1$ haben, $NNR(7.4) = 0$. Hier sehen wir das zentrale Merkmal: *NNR generalisiert sehr wenig.* Der Vorteil, dass wir keine weiteren Annahmen in die Daten stecken, ist also auch zugleich ein Nachteil, denn das Modell ist sehr anfällig für Unregelmäßigkeiten in den Daten, da es sie 1 zu 1 übernimmt. Das kann in manchen Fällen gut sein, insbesondere wenn es viel Varianz in den Daten gibt; das lässt sich aber schwer apriori sagen.

4 Vorspiel 2: Höhere Dimensionen

4.1 Lineare Modelle: Matrizen etc.

Wie wir gesehen haben, kommt linearen Modellen kommt im maschinellen Lernen eine überragende Bedeutung zu, vor allem da wir lineare Modelle in einer geschlossenen Form optimieren können. V.a. aber sind lineare Modelle auch von entscheidender Bedeutung, um die Bedeutung neuronaler Netze zu verstehen. Bislang haben wir die Modelle hauptsächlich als Funktionen

$$M : \mathbb{R} \to \mathbb{R}$$

aufgefasst. Wir werden jetzt die allgemeinere Perspektive nehmen mit

$$M : \mathbb{R}^n \to \mathbb{R}^m$$

In diesem Fall ist es sehr praktisch, mit Matrizen zu arbeiten. Ein lineares Modell L ist ein Modell der Form (für Eingabe $\mathbf{x}$)

$$L(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{a} \tag{53}$$

wobei $\mathbf{M}$ eine Matrix ist und $\mathbf{a}$ ein Vektor. Hier brauchen wir Matrixmultiplikation: sei also $\mathbf{x}$ ein Vektor mit Länge n, d.h. wir haben n Komponenten/Parameter als Eingabe. Das bedeutet: $\mathbf{M}$ ist eine Matrix mit $m \times n$ Feldern, wobei wir m frei wählen können (d.h. es ist unabhängig von der Eingabe – siehe Diskussion oben); $\mathbf{a}$ ist dann ein Vektor mit m Komponenten, ebenso wie die Ausgabe ein Vektor ist mit m Komponenten.

Um ein konkretes Beispiel zu liefern: nehmen wir an,

$$\mathbf{x} = (x_1, x_2, x_3)$$

hat Länge 3; wir wollen eine Ausgabe der Länge 2, also setzen wir $m = 2$, und

$$\mathbf{M} = \begin{pmatrix} 2.3 & 1.2 & 4 \\ 1 & 3.7 & 10 \end{pmatrix} \tag{54}$$

$$\mathbf{a} = (5, 6.3) \tag{55}$$

Dann haben wir also:

$$L(\mathbf{x}) = (2.3x_1 + 1.2x_2 + 4x_3, 1x_1 + 3.7x_2 + 10x_3) + (5, 6.3) \tag{56}$$

$$= (2.3x_1 + 1.2x_2 + 4x_3 + 5, 1x_1 + 3.7x_2 + 10x_3 + 6.3) \tag{57}$$

Das bedeutet: wir haben ein schönes, mehrdimensionales lineares Modell, das sich sehr einfach als Matrix darstellen lässt. Wir bekommen also als Ausgabe einen Vektor der Länge 2, wobei jede Komponente die drei Eingaben mit komponenten-spezifischen Parametern berücksichtigt. Das entscheidende ist: es bleibt bei den linearen Operationen Addition und Multiplikation.

Lineare Modelle haben eine Reihe von wichtigen Eigenschaften:

1. Sie sind geschlossen unter Komposition; d.h.: falls L_1 ein lineares Modell ist, L_2 ebenso und die Anzahl der Parameter stimmt, dann ist auch $L_1 \circ L_2$ ein lineares Modell, definiert durch

   (58) $L_1 \circ L_2(\mathbf{x}) = L_1(L_2(\mathbf{x}))$

2. Wenn wir Gleichung 56 betrachten, dann können wir mit elementaren Methoden die optimale Lösung finden. Das bedeutet: das Optimierungsproblem ist lösbar, wir finden immer die beste Lösung.

5 Nichtlineare Funktionen: Aktivierung etc.

5.1 Überblick

Die wichtigste Klasse von Funktionen sind die sog. Aktivierungsfunktionen. Wenn wir beispielsweise ein lineares Modell haben und unser eigentliches Ziel ist die Klassifikation, dann können wir mit der Ausgabe erstmal wenig anfangen – sie hat ja keinen offensichtlichen Bezug zu unserer Ausgabe. Aber selbst bei Regressionen haben wir oft das Problem: die Korrespondenz

$$\text{Größe der Matrix} \cong \text{Anzahl der Parameter}$$

bedeutet, die Länge unseres Ausgabevektors hängt in erster Linie ab von der Komplexität der Funktion die wir lernen sollen, nicht von der Art der Ausgabe, die wir suchen. Es kann sein dass unsere Funktion sehr komplex ist, aber am Ende suchen wir

$$M(\vec{x}) \in \mathbb{R}$$

Das bedeutet wir suchen normalerweise Aktivierungsfunktionen

(59) $A : \mathbb{R}^n \to \mathbb{R}$

Wenn wir klassifizieren möchten, dann möchten wir eine **konditionale Wahrscheinlichkeitsverteilung** bekommen, also eine Funktion

(60) $P(y|\vec{z}) \mapsto [0, 1]$

wobei $y \in Y$ eine endliche Menge ist, und $\mathbf{z}$ die Eingabe. Das bekommen wir in neuronalen Netzen, indem wir die mehrdimensionalen Funktionen mit A komponieren:

(61) $P(y|\mathbf{z}) = A \circ L(\vec{z})$,

wobei

(62) $L(\vec{z}) = \vec{x}$

der output der vorigen Funktion ist (z.B. ein lineares Modell). Das bedeutet, wir haben eine Funktion

(63) $P : Y \times \mathbb{R}^n \to [0, 1]$

so dass (im Idealfall)

$$\sum_{y \in Y} P(y, \mathbf{x}) = 1 \tag{64}$$

f.a. $\mathbf{x} \in \mathbb{R}^n$. Das bringt uns also auf die zwei Arten von nichtlinearen Funktionen:

1. Funktionen $f : \mathbb{R}^n \to \mathbb{R}^n$, die als "interne" Funktionen Parameter weiterreichen zwischen linearen Modellen

2. Funktionen $A : \mathbb{R}^n \to \mathbb{R}$, die idealerweise eine intuitive Bedeutung haben, z.B. als Wahrscheinlichkeit einer Klassifizierung.

5.2 Innere Funktionen (Aktivierungsfunktionen)

NB: diese Funktionen sind meistens Funktionen $f : \mathbb{R} \to \mathbb{R}$. Ihre Anwendung erfolgt dann auf Vektoren **komponentenweise**

Hier gibt es eine Funktion die man besonders hervorheben muss, nämlich die sog. **Sigmoid-Funktionen**. Eine besondere ist die sog. **logistische Funktion**

$$S(x) = \frac{1}{1 + e^{-x}} \tag{65}$$

Einige Beobachtungen: wir haben

- $S(0.5) = 0$
- $lim_{x \to -\infty} S(x) = 0$
- $lim_{x \to \infty} S(x) = 1$
- $S(x) \in (0, 1)$ f.a. $x \in \mathbb{R}$
- $S(x)$ ist streng monoton steigend in x

Allgemeiner nennt man solche Funktionen **Sigmoidfunktionen**. Ein anderes Beispiel für eine solche Funktion ist

$$T(x) = \frac{x}{1 + |x|} \tag{66}$$

$$U(x) = \frac{x}{1 + x^2} \tag{67}$$

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 - \frac{2}{e^{2x} + 1} \tag{68}$$

Was allen diesen Funktionen gemeinsam ist, ist dass sie im Bereich um 0 relativ schnell von 0 Richtung 1 wechseln, sonst für große positive/negative Zahlen langsam Richtung 1/0 konvergieren. Es gibt also nur einen kleinen Bereich, in dem eine nicht extreme Änderung im Argument eine signifikante Änderung im Wert verursacht; ansonsten gibt es relativ schnell eine Sättigung. solche Prozesse findet man oft in der Natur, z.B. bei Populationen. Solche logistischen Funktionen haben allerdings die Einschränkung dass sie nur $\mathbb{R} \to \mathbb{R}$ definiert sind. Aktivierungsfunktionen spielen in neuronalen Netzen eine große Rolle, und werden normalerweise auf die einzelnen Komponenten von Vektoren angewendet.

5.3 Punktweise Erweiterung von Funktionen

Wichtig ist: unsere linearen Modelle höherer Dimension liefern als Ausgabe eine Vektor, keine Zahl. Es gibt verschiedene Möglichkeiten, wie man die obigen nichtlinearen Funktionen auf Vektoren verallgemeinern kann, die einfachste ist: wir erweitern sie **punktweise**. Das geht wie folgt: Nehmen wir an, wir haben eine Funktion

$$f : \mathbb{R} \to \mathbb{R}$$

Wir können das erweitern auf beliebige

$$f : \mathbb{R}^m \to \mathbb{R}^m$$

mittels der Gleichung

$$f(x_1, ..., x_m) = (f(x_1), ..., f(x_m)) \quad (69)$$

Wir applizieren die Funktion also punktweise in jeder Dimension unabhängig voneinander. Das bedeutet wir bleiben in der gleichen Dimensionalität und ändern nur die Werte. Diese Methode wendet man oft an in den inneren Schichten (*hidden layers* der neuronalen Netze, denn hier möchten wir die Dimensionalität beibehalten. Anders sieht es oft bei Ausgabefunktionen aus: hier können wir mit Vektoren nicht soviel anfangen wie mit einfachen reellen Zahlen

5.4 Rectified linear units und ihre Generalisierung

RLUs sind definiert durch:

(70) $g(x) = max\{0, x\}$

Diese Funktion ist nicht linear und natürlich nur dann interessant, wenn man sie zusammen mit einer anderen, linearen Funktion verwendet. Wir haben also eine Funktion:

(71) $h(\mathbf{x}) = g(M\mathbf{x} + \mathbf{b})$

Das gute an diesen Funktionen ist natürlich ihre Einfachheit. Das Problem ist: sie haben ein langes Plateau, das ist Problem für Optimierung!

Es gibt 2 Generalisierungen von RLUs:

1. Es gibt eine (parametrisierte) Funktion die RLUs generalisiert:

   (72) $g(\vec{x}, \alpha)_i = max(0, x_i) + \alpha_i min(0, x_i)$

   Im Fall von $\alpha_i = 0$ haben wir eine RLU-Funktion. Falls $\alpha_i = -1$, dann wird das ganze zur Betragsfunktion, die man in diesem Fall **absolute value rectification** nennt. Man kann auch α_i zu einem kleinen Wert wie 1^{-n} setzen, mit z.B. $n = 100$; das ergibt **leaky** RLU. In **parametrischen** RLU ist α_i ein Parameter, der "gelernt" wird.

2. Es gibt noch die sog. **maxout units**. Anstatt komponentenweise eine Funktion zu applizieren werden die n Komponenten in Gruppen zu $k < n$ Werten aufgeteilt. Wir nennen diese Gruppe $G_1, ..., G_k$, wobei also

   $$G_1, ..., G_j \in \mathbb{R}^k$$

   Die Ausgabe der maxout Funktion ist wiederum ein Vektor, wobei

   (73) $g : \mathbb{R}^n \to \mathbb{R}^k$

   die Funktion für jede Gruppe eine Komponente ausgibt. Sie ist dementsprechend einfach definiert:

   (74) $g(\vec{z})_i = max(z_j : j \in G_i)$

Dadurch, das wir Gruppen bilden, ist die Funktion etwas robuster und weniger anfällig für Störungen in einzelnen Parametern.

5.5 Ausgabefunktionen

Hier suchen wir Funktionen

$$f : \mathbb{R}^m \to \mathbb{R}$$

Wir haben bereits gesehen, dass Normen diese Funktion erfüllen, aber das sind lineare Funktion und hier nicht von Interesse. Eine sehr wichtige Funktion ist die sog. *max*-Funktion, die einfach das Maximum ausgibt:

$$max(x_1, ..., x_m) = max\{x_1, ..., x_m\} \tag{75}$$

Sie liefert also den höchsten Wert. Das Problem dieser Funktion ist: sie ist nicht differenzierbar. Daher gibt es die sog. *softmax*-Funktion, die eine ähnliche Funktion übernimmt, aber differenzierbar ist (daher *soft*, weil sie keine Knicke hat).

$$softmax(x_1, ..., x_n)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{76}$$

Diese Notation ist gewählt um die Definition übersichtlicher zu machen; falls $\mathbf{x} \in \mathbb{R}^n$, dann ist auch $softmax(\mathbf{x}) \in \mathbb{R}^n$. Die softmax-Funktion ist eine Art "weiche" argmax-Funktion: dadurch dass wir exponentieren bekommen alle Komponenten, die nicht unbedeutend kleiner sind als der Größte, eine vernachlässigbare Größe. Der große Vortei von softmax ist: es ist eine kontinuierliche Funktion. Das ist sehr wichtig für die Optimierung, für die wir Gradienten brauchen. Softmax-Funktionen werden üblicherweise als Ausgabefunktionen verwendet: sie liefern tatsächlich eine Wahrscheinlichkeitsverteilung wie in (64).

Wichtig ist in allen diesen Funktionen, dass wir niemals ein Plateau haben, also z.B. niemals 0 erreicht für eine Sigmoidfunktion. Denn in diesem Fall können wir nicht weiter optimieren, der Gradient gibt uns keinerlei Information.

6 Tiefe Architekturen

Es gibt zwei Fragen, dir durchaus unabhängig voneinander behandelt werden können, nämlich:

1. Welche Funktionen können wir mit neuronalen Netzen repräsentieren?
2. Welche Funktionen können wir mittels Datensätzen "lernen"?

Um das zu sehen, bedenke man folgendes: es kann gut sein, dass wir eine bestimmte Funktion f mit einem Netz N mit i Schichten berechnen können, aber egal mit welchen Daten wir ein zufällig initialisiertes Netz mit i Schichten trainieren, es kommt niemals das Netz heraus, das f berechnet. Man muss also die beiden Probleme separat behandeln.

In diesem Abschnitt geht es also erstmal darum, welche Art von Funktion neuronale Netze überhaupt berechnen. Neuronale Netze sind im Prinzip nur eine Art, eine Klasse von Funktionen zu schreiben, die vielleicht am anschaulichsten ist, aber nicht um bedingt am besten geeignet, die Funktion zu verstehen und zu berechnen. Neuronale Netze versteht man am besten, wenn man sie als eine Abfolge von Schichten auffasst, wobei am Ende jede Schicht nacheinander appliziert wird (wir schauen uns das später im Detail an).

Eine Netz der Tiefe 1 berechnet folgende Funktion: sei $\mathbf{a}$ der Eingabevektor, M eine Matrix, $\mathbf{b}$ ein Vektor der passenden Länge, und g eine (nichtlineare) Aktivierungsfunktion.

Ein Netz N der Tiefe 1 berechnet nun die Funktion

$$N(\vec{x}) = g(M\vec{x} + \vec{b}) \tag{77}$$

Nun nehmen wir an, wir haben ein Netz N mit Schichten $N_1, N_2, ..., N_i$. Dann berechnet N die Funktion

$$N(\vec{x}) = N_i \circ ... \circ N_1(\vec{x}) = g_i(M_i(...g_1(M_1\vec{x} + \vec{b_1})...) + \vec{b_i}) \tag{78}$$

Das heißt: wir applizieren nacheinander die einzelnen Schichten. Wichtig ist hierbei: da lineare Funktionen unter Komposition geschlossen sind, würden die Schichten zu einer einzigen kollabieren, *wenn es nicht dazwischen die nonlinearen Funktionen gäbe*. Die Mächtigkeit von tiefen Netzen besteht also in der Alternation von linearen und nicht-linearen Schichten.

Parameter und Freiheitsgrade Hierbei ist folgendes wichtig: der Parameter, der die Anzahl der Zeilen der inneren Matrizen festlegt, hat erstmal keinerlei Bedeutung für die Natur der Ausgabe, sondern ist vielmehr wichtig für die *inneren* Berechnungen. Hierfür müssen wir nochmal anschauen, was Matrizen machen: wenn wir

$$M \in \mathbb{R}^{m \times n}$$

auf einen Vektor $\vec{x} \in \mathbb{R}^n$ applizieren, dann haben wir m *unabhängige* lineare Funktionen auf $\vec{x}$ appliziert, und die m Ergebnisse liefern uns den Ausgabevektor

$$M\vec{x} \in \mathbb{R}^m$$

Das bedeutet: je größer unsere Matrizen, desto mehr unabhängige lineare Operationen können wir ausführen, und desto mächtiger werden unsere Rechenmöglichkeiten. Man nenn daher den Parameter m auch den **Freiheitsgrad** des Models. Wir haben also ein Model

$$(\vec{x} \in \mathbb{R}^n) \overset{(M_1 \in \mathbb{R}^{m_1 \times n})}{\longrightarrow} (\vec{y}_1 \in \mathbb{R}^{m_1}) \xrightarrow{g_1} (\vec{y}_2 \in \mathbb{R}^{m_2}) ....$$

Hier sind m_1, m_2 etc. die Freiheitsgrade der Funktion und Hyperparameter, die nicht durch externe Datenformate vorgegeben sind, sondern empirisch optiminert werden müssen. Allgemein gilt auch hier:

Je kleiner m_1, m_2 etc., desto enger werden unsere Generalisierung gezogen ($\cong$ weniger Gefahr von *overfitting*), aber auch: desto weniger mächtig sind unsere Modelle ($\cong$ mehr Gefahr von *underfitting*)

7 Universelle Approximation

7.1 Mächtigkeit ist nicht Lernbarkeit

Wir kommen jetzt zum sog. universal approximation theorem, dass besagt, dass wir jede berechenbare Funktion mit einem Netzwerk approximieren können. Approximieren besagt hier soviel wie: die Differenz der beiden Funktionen wird vernachlässigbar klein. Das ist natürlich ein sehr wichtiges Ergebnis, denn es besagt dass unsere Netze praktisch universell sind, also alle möglichen Funktionen darstellen können.

Satz 1 *Für jede Borel-messbare Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$, jedes $\epsilon > 0$, gibt es ein neuronales Netz N mit einem hidden layer, so dass für alle $\vec{z} \in \mathbb{R}^n$ gilt:*

$$\|f(\vec{z}) - N(\vec{z})\| \leq \epsilon$$

Eine Funktion f ist Borel-messbar (etwas vereinfacht), falls das Urbild einer abzählbaren Vereinigung von geschlossenen Intervallen wiederum eine abzählbare Vereinigung geschlossener Intervalle ist.

Das besagt also: wir können bereits mit einem hidden layer *jede* solche Funktion beliebig approximieren! Allerdings ist dieses Ergebnis mit großer Vorsicht zu genießen: denn es sagt uns zwar etwas über Mächtigkeit, aber nichts darüber, wie wir die Funktionen *lernen* können. Zweitens sagt uns der Satz etwas über die Zahl der hidden layer, aber nichts über deren Größe: in der Tat sagt ein wichtiges Ergebnis, dass wir zwar die Zahl der hidden layer immer reduzieren können (bis auf 1), aber die Größe der layer wächst dabei exponentiell – das Vorgehen verbietet sich also in der Praxis.

Es gibt also insbesondere zwei Dinge zu unterscheiden:

1. Was Klassen von Funktionen *können*, und
2. wie Funktionen lernen.

Tiefe Netze sind insbesondere wichtig, denn je tiefer das Netz, desto besser sind die Generalisierungen, die Netze ziehen (für komplexe Aufgaben).

Man kann das sich wie folgt klarmachen: nehmen wir an, wir möchten eine Boolesche Funktion

$$B : \{0,1\}^n \to \{0,1\}$$

Es gibt viele solche Funktionen, nämlich

$$2^{2^n}$$

Wenn wir nun eine Funktion mit einem neuronalen Netz lernen möchten, das *ein* hidden layer hat, dann brauchen wir mindestens 2^n Zellen, und es ist schwierig, damit aus den Eingaben eine relevante Generalisierung zu ziehen. Das sieht ganz anders aus, wenn wir mehrere Schichten haben: dann kann z.B. Boolesche Entscheidungsbäume (bis zu einer gewissen Tiefe) simulieren und damit sehr schöne Generalisierungen treffen.

7.2 Ein Beispiel

Boolesche Funktionen kann man sehr schön nutzen, um die Art und Weise, wie neuronale Netze funktionieren, zu illustrieren. Nehmen wir die bekannte XOR-Funktion. Unsere Eingaben sind Vektoren in $\{0,1\}^2$. Wir machen nun die erste (lineare) Schicht:

$$M_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}$$

Damit bekommen wir:

$$M_1(1,0) = (1,0,-1,0) \tag{79}$$
$$M_1(0,1) = (0,1,0,-1) \tag{80}$$
$$M_1(1,1) = (1,1,-1,-1) \tag{81}$$
$$M_1(0,0) = (0,0,0,0) \tag{82}$$

Das transformiert also die Eingabe. Wir können nun eine *maxout*-unit als nichtlineare Transformation nehmen, mit $G_1 = \{1,2\}$, $G_2 = \{3,4\}$. So bekommen wir:

$$g \circ M_1(1,0) = (1,0) \tag{83}$$
$$g \circ M_1(0,1) = (1,0) \tag{84}$$
$$g \circ M_1(1,1) = (1,-1) \tag{85}$$
$$g \circ M_1(0,0) = (0,0) \tag{86}$$

Es ist nicht schwer zu sehen, dass die Vektoren nun linear separierbar sind entlang der gewünschten Grenze. Das sagt uns, dass wir bereits mit einem linearen Modell auskommen. Wir nehmen eine sehr einfache Matrix:

$$M_2 = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

Natürlich gilt hier einfach: $M_2(x_1, x_2) = x_1 + x_2$, also:

$$M_2 \circ g \circ M_1(1,0) = 1 \tag{87}$$

$$M_2 \circ g \circ M_1(0,1) = 1 \tag{88}$$

$$M_2 \circ g \circ M_1(1,1) = 0 \tag{89}$$

$$M_2 \circ g \circ M_1(0,0) = 0 \tag{90}$$

An dieser Stelle ist eine weitere nicht-lineare Funktion unnötig, unser Netz berechnet bereits die XOR-Funktion.

7.3 Allgemeiner: Entscheidungsbäume

Jede Boolesche Funktion

$$B : \{0,1\}^n \to \{0,1\}$$

lässt sich mittels eines binären Entscheidungsbaumes mit Tiefe $\leq n$ darstellen. Der Witz an dieser Darstellung ist, dass wir oftmals nicht den vollen Entscheidungsbaum der Tiefe n brauchen, sondern nur ausnahmsweise, z.B. für Funktionen wie

$$B_{ger}(a_1, ..., a_n) = 1 \Leftrightarrow a_1 + ... + a_n \text{ gerade} \tag{91}$$

Sonst reicht es oftmals, eine bestimmte Kombination von z.B. 3 Merkmalen zu kennen (aus z.B. 20) um bereits eine Entscheidung zu treffen. Die Konstruktion von guten Entscheidungsbäumen basiert dabei darauf, dass wir die wichtigsten/informativsten Merkmale zuoberst setzen, und die unwichtigsten zuunterst. Hiermit können wir insbesondere in der Praxis gute Generalisierungen treffen. Wohlgemerkt: das muss nicht so sein, ist es aber in der Praxis, hängt also ab von der Klasse von Problemen, die wir betrachten.

Diese Beobachtung kann man nun auf neuronale Netze übertragen: ein Netz mit einem hidden layer wird als solches keine Boolesche Funktion vor einer anderen bevorzugen: erstmal stehen alle auf derselben Stufe. Wenn wir ein Netz mit n hidden layern haben, dann können wir uns das vorstellen als einen (viel mächtigeren) Entscheidungsbaum mit n Ebenen. Mächtiger, denn wir können auf jeder Ebene viel mehr machen als ein Entscheidungsbaum, aber: jede Generalisierung des Entscheidungsbaumes lässt sich eben auch auf

der Ebene eines layers fassen. Wenn nun ein Merkmal M_i besonders wichtig ist, dann kann also der Unterschied zwischen

$$B_{M_i=0} \tag{92}$$

$$B_{M_i=1} \tag{93}$$

sehr weit oben im Netz gezogen werden. Nimm hierzu die Eingabe

$$\vec{x} = (x_1, ..., x_{i-1}, 0, x_{i+1}, ...x_n) \tag{94}$$

und das lineare Modell mit

$$M = \begin{pmatrix} x_{1,1} & x_{1,2} & ...x_{1,i} & .... & x_{1,n} \\ .... \end{pmatrix} \tag{95}$$

wobei wir setzen:

$$x_{1,1} = x_{1,2} = ... = x_{1,i-1} = x_{1,i+1} = ... = x_{1,n} = 0 \tag{96}$$

$$x_{1,i} = 1 \tag{97}$$

In diesem Fall wird der gesamte Wert der ersten Ausgabezeile 0 gdw. $x_i = 0$, während die restlichen Zeilen andere Merkmale filtern. Eine nichtlineare *maxout*-unit erlaubt uns, verschiedene Gruppen in Abhängigkeit zu setzen, z.B. durch

$$\vec{z_j} = max(y_1, y_k) \tag{98}$$

Hier bekommen wir also z.B. den Wert 1, falls $x_{1,i} = 1$ (je nach herangehensweise). Die nächst Schicht prüft das zweitwichtigste Merkmal etc.

Der Trick ist hier: dadurch, dass wir Berechnungen, die wir auch in einem layer durchführen könnten, auf verschiedene layer verteilen, führt unser Lernalgorithmus zur schnelleren Optimierung (gradient descent) gdw. wir richtig generalisieren. Eine starke Berücksichtigung des wichtigsten Merkmals im ersten layer wird schneller die Ergebnisse verbessern, und daher wird es eher erfolgen (wg. Gradienten!). Hier sehen wir die Herausforderung für Lernalgorithmen in tiefen Netzen: wir wollen zuerst allgemeinere Generalisierungen in früheren Schichten ziehen. Es ist aber mathematisch nicht garantiert, dass das einfach funktioniert, es ist nur konzeptuell einleuchtend!

Das führt uns also zum schwierigen Problem von Training und Optimierung neuronaler Netze.

8 Optimierung

8.1 Vorspiel 1: Analytische Optimierung

Bei linearen Modellen haben wir eine sog. analytische Optimierung, das bedeutet: wir finden mit den Methoden der Analysis (Ableitungen etc.) in einem Schritt die optimale Lösung. Das funktioniert bei linearen Modellen (ein oder mehrdimensional) und nearest neighbour regression, also den sehr einfachen Modellen. Bei komplexeren, nicht-linearen Modellen funktioniert das nicht mehr: wir müssen uns schrittweise an eine besseren Lösung herantasten. Oftmals ist es auch nicht klar, dass wir überhaupt eine optimale Lösung finden: es kann sein dass wir in einem lokalen Minimum bleiben.

8.2 Vorspiel 2: Konvexe Optimierung

Konvexe Optimierungsprobleme sind ein Spezialfall, der deswegen interessant weil er besonders gut gelöst werden kann. Eine **konvexe Menge** $X \subseteq \mathbb{R}$ ist eine Menge, die folgende Bedingung erfüllt:

$$\text{Falls } x, y \in X,\ x < z < y, \text{ dann ist } z \in X.$$

Allgemeiner:

Definition 2 *Eine Menge $X \subseteq \mathbb{R}^n$ ist konvex, falls gilt: für alle Skalare $\lambda \in [0,1]$, $\vec{x}, \vec{y} \in X$, $\lambda x + (1-\lambda)y \in X$.*

Diese Definition ist erstmal nicht intuitiv, beruht aber auf der Parameterdarstellung von Verbindungsstrecken zweier Punkte:

(99) $\overline{xy} = \{\lambda x + (1-\lambda)y : \lambda \in [0,1]\}$

ist die Menge aller Punkte, die auf der geraden Verbindung von x, y liegen. D.h. wir haben hier ein geometrisch sehr intuitives Konzept. Eine **Funktion**

$$f : \mathbb{R}^m \to \mathbb{R}$$

ist konvex, falls

$$\uparrow f = \{(x_1, ..., x_n, y) : f(x_1, ..., x_n) \leq y\}$$

eine konvexe Menge ist. Konvexe Funktionen haben also einen Funktionsgraphen, der eine konvexe Menge nach unten begrenzt, anders gesagt: die Menge der Punkt die über dem Graphen liegen ist konvex. Die großen Vorteile konvexer Funktionen sind folgende:

- Jedes lokale Minimum der Funktion ist ein globales Minimum (vorausgesetzt es existiert eines).
- Die Menge der globalen Minima (sofern es mehr als eines gibt) ist wiederum konvex, d.h. sie liegen beieinander.

Das bedeutet natürlich einen enormen Vorteil: um

(100) $argmin f(x)$

zu finden, müssen wir nur prüfen ob

$$f(x+\epsilon) < f(x) \text{ oder } f(x-\epsilon) < f(x),$$

und irgendwann werden wir zwangsläufig die optimale Lösung treffen. Insbesondere gilt: falls

(101) $f(x) > f(x+\epsilon) < f(x+2\epsilon)$

dann ist $argmin(f) \in (x, x+2\epsilon)$.

Konvexe Optimierungsprobleme sind wie folgt definiert: Ein Optimierungsproblem besteht darin, für $X \subseteq \mathbb{R}^n$ und $f : \mathbb{R}^n \to \mathbb{R}$

(102) $argmin_{x \in X} f(x)$

zu finden. Das Optmierungsproblem ist ist **konvex**, falls

1. f eine konvexe Funktion ist, und
2. X eine konvexe Menge.

Alternativ und äquivalent kann man das Optimierungsproblem so beschreiben: wir haben eine Reihe von Funktionen

$$f, g_1, ..., g_n : \mathbb{R}^n \to \mathbb{R},$$

und wir suchen

(103) $argmin_{x \in \mathbb{R}^n} f(x)$, vorausgesetzt dass $g_1(x), g_2(x), ..., g_i(x) \leq 0$

Das Optimierungsproblem ist **konvex**, falls $f, g_1, ..., g_i$ konvexe Funktionen sind.

9 Gradienten berechnen

Um Gradienten zu berechnen braucht man einfach nur die partielle Ableitung. Der Begriff der partiellen Ableitung generalisiert den der Ableitung zu multivariaten Funktionen:

$$\frac{df}{dx_i} f(x_1, ..., x_i, ..., x_n) = \frac{df}{dx} f(a_1, ..., x, ..., a_n) \tag{104}$$

wobei $a_1, ..., a_n$ einfach als Konstanten (d.h. wie einfache Zahlen behandelt werden). Z.B. haben wir:

$$\frac{df}{dx} x^2 + 2y = 2x \tag{105}$$

und

$$\frac{df}{dy} x^2 + 2y = 2 \tag{106}$$

Allerdings gilt:

$$\frac{df}{dx} x^2 \cdot 2y = 2x \cdot 2y \tag{107}$$

und

$$\frac{df}{dy} x^2 \cdot 2y = 2x^2 \tag{108}$$

Wir müssen also die Variablen, nach der nicht abgeleitet wird, einfach als normale Konstante lesen; auf diese Art und Weise kommen von der partiellen zu einer normalen Ableitung. Da alle Variablen (bis auf evtl. diejenige, nach der abgeleitet wird) in der Funktion weiterhin vorkommen, haben wir eine numerische Funktion an einem gewissen Punkt.

Wenn wir die partielle Ableitung über alle Variablen berechnen, bekommen wir den Gradienten.

10 Gradientenbasierte Optimierung

10.1 Das Problem in höheren Dimensionen

Wenn wir schrittweise – also nicht analytisch – optimieren, dann macht es einen grossen Unterschied ob wir eine Funktion $f : \mathbb{R} \to \mathbb{R}$ haben, also nur einen Parameter optimieren, oder eine Funktion $f : \mathbb{R}^n \to \mathbb{R}$ haben, wo wir also

$$\underset{\vec{x}\in\mathbb{R}^n}{argmin} f(\vec{x}) \tag{109}$$

suchen. Der Grund ist folgender: wenn wir

$$\underset{x\in\mathbb{R}}{argmin} f(x) \tag{110}$$

suchen, dann reicht es oft aus, wenn wir an einem (mehr oder weniger zufällig gewählten Punkt $a \in \mathbb{R}$ anfangen, dann a ersetzen durch $a + l$ und prüfen ob

$$f(a) > f(a + l) \tag{111}$$

(l ist hierbei die Lernrate) Falls ja, prüfen wir ob

$$f(a + l) > f(a + l + l) \tag{112}$$

etc. Wenn wir bei einem Vektor $\vec{a}$ anfangen, ist die Sache viel komplizierter: wir können $\vec{a}$ nicht nur mit verschiedenen Lernraten ändern, sondern auch in *unendlich viele Richtungen* (statt nur zwei). Wir müssen also erstmal eine Richtung finden, in die wir das Argument ändern müssen, d.h. einen Vektor $\vec{b}$, für den wir prüfen ob

$$f(\vec{a}) > f(\vec{a} + \vec{b}l) \tag{113}$$

wobei l (ein Skalar) weiterhin die Lernrate ist, die bestimmt wie weit wir uns in eine Richtung begeben.

Man ermittelt $\vec{b}$ mittels des Gradienten:

$$\vec{b} \Leftarrow \nabla f(\vec{a})$$

ist ein Vektor, und dieser Vektor zeigt an, in welche Richtung der Wert von f am Punkt $\vec{a}$ am steilsten steigt (bzw. fällt). Also ist die neue Methode: wir berechnen $\nabla(f)$, und prüfen ob

(114) $f(\vec{a}) < f(\vec{a} + \nabla f(\vec{a})l)$

wobei l wie vorher die Lernrate ist, und $\nabla f(\vec{a})$ die Richtung angibt, in die wir explorativ den Wert verändern. Wohlgemerkt:

- Der Gradient gibt uns den steilsten Anstieg für eine infinitesimale Veränderung, d.h. es ist nicht gesagt das $f(\vec{a}) < f(\vec{a} + \nabla f(\vec{a})l)$. Es ist nur gesagt dass es ein $\epsilon > 0$ gibt, so dass $f(\vec{a}) < f(\vec{a} + \nabla f(\vec{a})\epsilon)$. Das ϵ kennen wir aber nicht!
- Es ist auch nicht gesagt, dass diese Methode ein optimales Ergebnis liefert: wir können ja in einem lokalen Minimum gefangen werden.
- Aber: wenn wir die Lernrate l sukzessive verringern mit $l_1, l_2, ...$ mit $lim_{n\to\infty} l_n = 0$, dann werden wir ein lokales Minimum finden!

10.2 Die effektive Prozedur

Bei gradientenbasierter Optimierung läuft es daraus hinaus, dass wir maschinelles Lernen als klassisches Optimierungsproblem auffassen, d.h. wir suchen

(115) $argmin_{x_1,...,x_n \in \mathbb{R}^n} f(x_1, ..., x_n)$

gegeben dass

(116) $$g_1(x_1, ..., x_n) < 0$$

(117) $$...$$

(118) $$g_i(x_1, ..., x_n) < 0$$

Hier sind g_1 bis g_n die Parameter, währen f die Funktion ist die optimiert werden soll. Im multivariaten Fall kann man nicht mehr einfach "die" Ableitung bilden und auf 0 setzen – es gibt ja viele Ableitungen. Stattdessen muss der Gradient den Wert $(0, ..., 0)$ haben an einem Punkt. Nullstellen für Gradienten lassen sich aber allgemein nicht berechnen. Im Normalfall können wir die Funktion also nur schrittweise minimieren, d.h.: ihren Wert verringern. Zu diesen Zweck legen wir eine **Lernrate** l fest.

Wir fangen an einem beliebigen Punkt $\vec{a} = (a_1, ..., a_n) \in \mathbb{R}^n$ an; der Punkt wird zunächst mehr oder weniger zufällig ausgewählt. Als nächstes nehmen wir ∇f, und setzen:

(119) $\vec{a}' := \vec{a} + l\nabla f(\vec{a})$

Das lässt sich wie folgt verstehen: $\nabla f(\vec{a})$ zeigt uns die Richtung an, in der die Funktion vom Punkt $\vec{a}$ *am steilsten* abfällt. $l\nabla f(\vec{a})$ ist nun das Skalarprodukt mit l, der Lernrate. Was 120 also besagt ist: wir bewegen uns vom Punkt $\vec{a}$ in Richtung des steilsten Gefälles, die Wegstrecke, die wir zurücklegen ist dabei l, und der Punkt an dem wir anlangen ist $\vec{a}'$. Nun wiederholen wir das ganze:

(120) $\vec{a}'' := \vec{a}' + l\nabla f(\vec{a'})$

etc. bis wir eine zufriedenstellende Lösung haben, oder aber im unwahrscheinlichen Fall dass wir tatsächlich ein Minimum erreicht haben.

Hier gibt es einiges zu beachten: die Tatsache, dass wir Entfernung l in Richtung des steilsten Gefälles zurücklegen sagt *nicht*, dass wir den tiefsten Punkt im Abstand l erreichen – das ist sogar recht unwahrscheinlich im

allgemeinen Fall. Und selbst wenn wir den niedrigsten Punkt im exakten Umkreis l finden, muss das nicht der niedrigste Punkt im Umkreis $\leq l$ sein. Es kann sogar theoretisch sein, dass

$$\vec{a} = \vec{a}''$$

und wir in einer Art Schleife gefangen sind. Der Punkt ist: obwohl das theoretisch möglich, ist es in der Praxis so gut wie ausgeschlossen. D.h. wir haben ein Verfahren, das theoretisch nicht unbedingt gut funktioniert, in der Praxis aber schon.

Eine Grundvoraussetzung dafür, dass das Verfahren funktioniert, ist dass die Funktion f nicht übermäßig komplex ist. Z.B.: je weniger Wendepunkte die Funktion hat, desto eher finden wir zum globalen Minimum. In der Praxis gilt: wir finden nicht unbedingt ein lokales Minimum, aber normalerweise kommen wir in die Nähe. Andererseits gibt es keine Möglichkeit, lokale und globale Maxima zu unterscheiden. Daher sind wir auf gewissen Eigenschaften unserer Funktionen angewiesen, wenn wir gute Ergebnisse garantieren möchten. Eine wichtige Eigenschaft, die wir explizit besprechen werden, ist **Konvexität**: das bedeutet (unter anderem) dass jedes lokale Minimum ein globales Minimun ist.

Was ebenfalls ein großer Vorteil ist, ist falls

$$\nabla^n f = 0 \tag{121}$$

wobei n möglichst klein sein sollte. Damit können wir sicher gehen, dass wir nur endlich oft die Richtung (Auf- und Abstieg) wechseln.

11 Differenzieren – einige Faustregeln

Differenzierung ist vornehm für Ableitung; hier wollen wir einige Ableitungsregeln besprechen, damit wir verstehen was hinterher passiert. Zunächst folgende **Additionsregel**:

$$(122)\quad (f(x) + (g(x))' = f(x)' + g(x)'$$

Diese Regel gilt natürlich per Definition auch für partielle Ableitungen, da das einfach Ableitungen sind, die gewisse Variablen als Konstanten auffassen:

$$(123)\quad \frac{\partial f}{\partial x}(f(x,\vec{y}) + g(x,\vec{y}))' = \frac{\partial f}{\partial x}f(x,\vec{y}) + \frac{\partial f}{\partial x}g(x,\vec{y})$$

Das bedeutet, wir können die Additionsregel auf Gradienten erweitern, da Gradienten nur Tupel von partiellen Ableitung sind:

$$(124)\quad \nabla(f(\vec{x} + g(\vec{x})) = \nabla f(\vec{x}) + \nabla g(\vec{x})$$

Die nächste wichtige Regel ist die **Kettenregel**. Sie liefert uns die Ableitung einer Funktion $f \circ g$, definiert durch

$$(125)\quad f \circ g(x) = f(g(x))$$

Die Regel lautet:

$$(126)\quad (f \circ g(x))' = f'(g(x)) \cdot g'(x)$$

zum besseren Merken: innere Ableitung (g) mal äußere Ableitung (f). Z.B. wissen wir dass

$$(127)\quad (\frac{1}{x})' = -\frac{1}{x^2}$$

Wenn wir nun die Ableitung von $f(x) = \frac{1}{3x^2}$ berechnen wollen, geht das mittels

$$(128)\quad f(x) = \frac{1}{x} \circ 3x^2(x)$$

Also haben wir

$$(129)\quad f(x)' = -\frac{1}{(3x^2)^2} \cdot 6x = -\frac{6x}{9x^4} = -\frac{2}{3x^3}$$

Wo kommt diese Regel her? Erinnern wir uns das

$$f(x)' = \lim_{x \to 0} \frac{f(x)}{x} \tag{130}$$

die Funktion ist, die die Veränderung von $f(x)$ für eine *inifitesimale* Veränderung von x liefert. Die Kettenregel kann man damit sehr schön ableiten, indem man den Bruch einfach erweitert:

$$\lim_{x \to 0} \frac{g \circ f(x)}{x} = \lim_{x \to 0} \frac{g \circ f(x)}{f(x)} \frac{f(x)}{x} \tag{131}$$

Hier passiert also nichts als eine Erweiterung des Bruches. Nun ist aber

$$\lim_{x \to 0} \frac{g(x)}{f(x)} = g'(f(x)) \tag{132}$$

$$\lim_{x \to 0} \frac{f(x)}{x} = f'(x) \tag{133}$$

denn wir schauen, wie sich der Wert von $g \circ f$ ändert für eine infinitesimale Änderung von f ändert; der zweite Teil ist unmittelbar klar.

Was wir evtl. noch brauchen ist die Ableitung der Aktivierungsfunktionen, die nicht ganz trivial ist. Tatsächlich haben wir

$$\frac{df}{dx} \frac{1}{1 + e^{-x}} = \frac{e^x}{(e^x + 1)^2} = \frac{1}{1 + e^{-x}} \cdot (1 - \frac{1}{1 + e^{-x}}) \tag{134}$$

Kurz gesagt, wenn wir die Sigmoid-Funktion mit $S(x)$ abkürzen, dann bekommen wir

$$S(x)' = S(x) \cdot (1 - S(x)) \tag{135}$$

Diese Funktionen brauchen wir für Backpropagation, und das ist fürs erste alles was wir brauchen, um Backpropagation zu verstehen.

12 Tiefe Netze trainieren

12.1 Parameter und Hyperparameter

Beim Training von Netzen müssen wir erstmal unterscheiden, was die Parameter sind, die optimiert werden sollen, und was die Hyperparameter sind, die extern festgelegt sind. **Hyperparameter** sind:

- Die Anzahl der Layer.
- Die Natur der inneren nichtlinearen layer (RLU, maxout, Sigmoid,...)
- Die Größe der linearen layer (= Anzahl der Zeilen der Matrizen)
- Die Natur der Ausgabefunktion (softmax, max,...)

Parameter sind:

- Die Werte der Matrizen im linearen Layer.
- Je nach nichtlinearen layern: z.B. die Gruppen der maxout-unit, die Parameter für parametrisiertes RLU.

Wenn wir also tiefe Netze trainieren, dann bekommen wir folgendes Problem: wir haben wiederum gegeben:

1. $D \subseteq \mathbb{R}^n \times \mathbb{R}$, unsere Daten. Wir schreiben auch $D = \{(\vec{x}_1, y_1), ..., (\vec{x}_i, y_i)\}$
2. Unsere Kostenfunktion $K : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$.
3. Unsere Netzwerkarchitektur N mit gesetzten Hyperparametern. Auch unsere Parameter sind auf eine mehr oder weniger zufällige Art und Weise initialisiert.

Wir messen also

$$\sum_{j=1}^{i} K(N(\vec{x}_j), y_j) \tag{136}$$

als unsere Gesamtkosten; das ist der Term den wir minimieren möchten. Minimieren heißt: über die Parameter von N. Wir müssen das Modell also ausbuchstabieren:

$$N(\vec{x}) = g_{out} \circ M_k \circ g_{k-1} \circ M_{k-1} \circ ... \circ g_1 \circ M_1(\vec{x}) \tag{137}$$

Wir haben also eine sehr komplexe Aufgabe vor uns, wir suchen:

$$\underset{M_k, M_{k-1}, \ldots, M_1, g_{k-1}, \ldots, g_1}{argmin} \sum_{j=1}^{i} c(g_{out} \circ M_k \circ g_{k-1} \circ M_{k-1} \circ \ldots \circ g_1 \circ M_1(\vec{x}_j), y_j) \tag{138}$$

Es ist natürlich klar, dass man für 138 keine analytische Lösung findet, es ist bereits schwer genug, einen Gradienten zu finden. Dafür fässt man diese Funktion einfach als eine Funktion auf, in der jeder Parameter eine Variable ist (also sehr hochdimensional!). Diese Funktion ist natürlich fast unmöglich sinnvoll darzustellen; man muss dazu die Variablen der Matrizen etc. linearisieren, Gruppen in *maxout*-units numerisch darstellen etc.

Wichtig ist das Konzept: ein Funktion in vielen Variablen braucht einen Gradienten. Den Versuchen wir zu verringern.

12.2 Backpropagation und Stochastic gradient descent

Für das Training tiefer Netze gibt es zwei wichtige Techniken:

1. Backpropagation, ein Algorithmus um den Gradienten des Netzes in einem gewissen Punkt zu berechnen (Gradient der Funktion F im Punkt $\vec{x} \in \mathbb{R}^n$ ist $\nabla F(\vec{x}) \in \mathbb{R}^n$, wir suchen also einen Vektor!)

2. Stochastic gradient descent, eine Methode, mittels derer wir versuchen, den Gradienten auf unseren Datenpunkten zu minimieren. Das besteht aus zwei Teilen:

   (a) Eine Sampling-Technik: da unser Verfahren nicht analytisch ist, müssen wir unsere Datenpunkte, auf denen wir optimieren, immer wieder (mehr oder weniger) zufällig auswählen.

   (b) Backpropagation liefert uns den Gradienten über die *Eingabe* des Netzes, nicht der **Parameter**. Wir können aber das Netz so umstellen, dass die Parameter der ersten Schicht die Eingaben werden, und das Schrittweise fortführen.

Diese beiden Techniken ergänzen sich und sorgen jeweils für bestimmte Teilbereiche des Problems. Nehmen wir eine Kostenfunktion K an, die für jede Ausgabe des Netzes (auf den Trainingsdaten) die Kosten liefert, als Metrik zum Zielpunkt. Die Komposition der beiden nennen wir F (wie Fehler). Wir haben also eine Funktion

$$K \circ N(x) = F(x) \tag{139}$$

Wir gehen nun wie üblich vor: wir ändern die Funktion dahingehend, dass die Parameter von N die Variablen werden, x dagegen als fester Wert gesetzt wird. Diese Funktion nennen wir

$$K_x(N) \tag{140}$$

Wir suchen nun die Minimalstelle von $K_x(N)$, also dasjenige N, dass die Kosten minimiert. Das ist natürlich eine Funktion mit wahnsinnig vielen Variablen.

Wir lösen erstmal nur einen Zwischenschritt mittels Backpropagation: wir versuchen den Gradienten von F auf einem bestimmten Datenpunkt zu berechnen, um uns somit der Nullstelle der Funktion zu nähern. Da

$$F : \mathbb{R}^n \to \mathbb{R}_0^+ \tag{141}$$

eine positive Funktion ist (wegen K) ist das ausreichend um ein (lokales) Minimum zu finden.

Backpropagation ist ein Algorithmus, der dazu dient, den Gradienten zu berechnen; SGD eine (heuristische) Methode, die Parameter in Richtung eines (lokalen) Minimums zu ändern.

12.3 Die Fehlerfunktion

Wir haben die Fehlerfunktion F dargestellt. Diese Funktion hat viele Variablen, und in der Berechnung des Gradienten müssen wir nach jeder Variablen differenzieren. Vereinfacht sieht diese Funktion so aus, for our training data $\{(\vec{x}_i, \vec{y}_i) : i \in I\}$

$$F = \frac{1}{2} \sum_{i=1}^{n} \|N(\vec{x}_i - \vec{y}_i\|^2 \tag{142}$$

Wichtig ist hierbei, dass $F : \mathbb{R}^n \to \mathbb{R}_0^+$, wir also eine positive Funktion haben. Der Term $\frac{1}{2}$ wird nur Einfachheit halber gesetzt; in der Ableitung kürzt er sich weg.

Wir nehmen dabei erstmal an, dass die Aktivierungsfunktionen g feststehen, also nicht optimiert werden. Es geht also nur um die linearen Modelle. Hier ist $w_{i,j}$ alternativ der zugehörige Matrix-Eintrag, bzw. die Gewichtung von Eingabe i und Neuron j. (andersrum)

Wichtig ist: wir lassen unser Netz nun die Fehlerfunktion berechnen. Das geht ganz einfach, indem man am Ende eine Zelle hinzufügt, die K berechnet. Wir haben also ein Netz mit einer einzigen Ausgabe $o \in \mathbb{R}$!

12.4 Backpropagation Übersicht

Backpropagation ist ein Algorithmus, der als **Eingabe** nimmt:

1. ein initialisiertes Netz (mit gesetzten Parametern), Kostenfunktion
2. eine Eingabe $\vec{x}$,
3. eine (Gold-)Ausgabe $\vec{y}$, die wir benötigen um die Kostenfunktion zu instantiieren; so haben wir F

Was wir bekommen als **Ausgabe** ist ein (konkreter) Gradient, also:

$\Rightarrow \nabla F(\vec{x}) \in \mathbb{R}^n$, für $\vec{x} \in \mathbb{R}^n$

Wir werden jetzt den Algorithmus erklären, allerdings nicht mittels Pseudocode (dass wäre zu langwierig).

12.5 Berechnungsgraphen

Zunächst ist es sehr hilfreich, Berechnungsgraphen einzuführen. Diese sehen ähnlich aus wie neuronale Netze, aber etwas anders. In einem BG entspricht jeder Knoten einer atomaren Berechnung, in unserem Fall entweder

1. Addition verschiedener variabler Eingaben
2. Multiplikation einer variablen Eingabe mit einer Konstanten
3. Applikation einer nichtlineraren Funktion S auf eine variable Eingabe

Die Kanten bedeuten jeweils, dass die Ausgabe eines Knotens als Eingabe an einen anderen Knoten weitergereicht werden. Im neuronalen Netz hatten wir 2. an eine Kante geschrieben; stattdessen setzen wir es jetzt in eine eigenen Knoten. *Ein* Knoten im Netz hatten bislang sowohl 1. als auch 3. berechnet; auch das müssen wir jetzt aufdröseln.

GRAFIK!

12.6 Backpropagation – Verwaltung

Wichtig für BP ist die richtige Verwaltung der Knoten. Wir nutzen die Konvention:

- Für jeden Knoten haben wir einen Index $i \in \mathbb{N}$; wir bezeichnen den Knoten mit v_i.
- Für jeden Knoten haben wir eine Menge von Knoten, die eine Eingabe liefern, und eine Menge, die seine Ausgabe weiterverarbeiten. Die erste Menge bezeichnen wir mit $\uparrow v$, die zweite Menge mit $\downarrow v$. Das sind also die **unmittelbaren** Vorgänger und Nachfolger.
- Wir nutzen auch manchmal die transitive Hülle dieser Relationen; wir bezeichnen das mit $\uparrow^* v$ bzw. $\downarrow^* v$.
- Wir schreiben $\downarrow^1 v = \downarrow v$, und $\downarrow^{n+1} v = \downarrow\downarrow^n v$; ebenso für $\uparrow$.

Wichtig für die Verwaltung von Netzen ist folgende Tatsache:

- Gegeben einen beliebigen Knoten v im Berechnungsgraphen, haben wir mit $\uparrow^* v$ ein **Subnetz**, das aus allen Vorgängern besteht und eine einzelne Ausgabe $o \in \mathbb{R}$ hat.

Noch ein weiterer wichtiger Begriff ist der des **dichten Netzes**. Ein Netz ist dicht, falls folgendes gilt:

$$\text{Falls } (\uparrow^n v) \cap (\uparrow^n v') \neq \emptyset, \text{ dann ist } (\uparrow^n v) = (\uparrow^n v')$$
$$\text{Falls } (\downarrow^n v) \cap (\downarrow^n v') \neq \emptyset, \text{ dann ist } (\downarrow^n v) = (\downarrow^n v')$$

Das bedeutet: falls 2 Knoten einen gemeinsamen Nachfolger in einer gewissen Tiefe haben, dann sind alle Nachfolger gemeinsam. Die einfache Variante von Backpropagation funktioniert nur für dichte Netze.

Wir nehmen nun für jeden Knoten zwei Einträge. Erinnern wir uns, dass v einer Funktion f_v entspricht. Wir definieren nun:

- $r(v) = f_v$
- $l(v) = f'_v$

Das bedeutet rechts steht die ursprüngliche Funktion, links die Ableitung.

12.7 Backpropagation – Der Feedforward-Step

Wir arbeiten nun das Netz von vorne nach hinten um. Wir nehmen die Eingabe $\vec{x} = (x_1, ..., x_n)$ für den Berechnungsgraphen von F. Wir setzen nun sukzessive für alle Knoten folgendes ein (von vorne nach hinten):

Fall 0 Die Eingabeknoten
Der Eingabknoten v_i bekommt

$$r(v_i) = x_i \tag{143}$$

$$l(v_i) = 1 \tag{144}$$

Da $f_{v_i}(x) = x$ – der Eingabe-Knoten "berechnet" die Identität – ist der linke Term der Wert $f'_{v_i}(x)$.

Wir kommen jetzt zu den inneren Knoten:

Fall 1 Knoten mit einem Vorgänger
Wir haben v mit $|\uparrow v| = 1$

$$r(v) = f_v(r(\uparrow v)) \tag{145}$$

$$l(v) = f'_v(r(\uparrow v)) \tag{146}$$

D.h. wir speichern den aktuellen Wert der forward-Berechnung im Knoten v in $r(v)$; in $l(v)$ speichern wir die Ableitungsfunktion appliziert auf den Vorgänger.

Man beachte: falls v einer (konstanten) Multiplikation entspricht, dann ist $f'_v(x) = w$, also $l(v) = w$ – denn es kommt keine Variable darin vor!

Fall 2 : $f_v = x_1 + ... + x_i$
Der Knoten hat also eine Menge von Vorgängern.

$$r(v) = \sum_{v' \in \uparrow v} r(v') \tag{147}$$

$$l(v) = (1, ..., 1) \tag{148}$$

Auch das ist klar (auch wenn es hier der Gradient ist, nicht die Ableitung, denn $\nabla f_v = (1, ..., 1)$, d.h. $\nabla f_v(r(\uparrow v(x)) = (1, ..., 1)$ – wir haben eine Funktion mit einer konstanten Ausgabe.

Damit sind wir fertig; unsere Ausgabeeinheit v_o ist also derart, dass

$$r(v) = F(\vec{x}) \tag{149}$$

$$l(v) = (1, ..., 1) \tag{150}$$

denn die letzte Rechenoperation ist eine Addition! Wenn wir also im FF-Schritt unseren Graphen präparieren, dann funktioniert das nach dem einfachen Rezept: links Ableitung/Gradient auf Eingabe, rechts Funktion auf Eingabe. Das wars.

12.8 Backward-propagation

Im BP-Schritt lassen wir das ganze jetzt rückwärts laufen. Während wir vorher das Netz präpariert haben, und am Ende die Ausgabe o für Eingabe $\vec{x}$ bekommen haben, lassen wir die Berechnung jetzt von rechts nach links laufen, wobei am Ende wir eine Ausgabe $\vec{z}$ bekommen (an den alten Eingabeknoten), wobei gelten soll

(151) $\vec{z} = \nabla F(\vec{x})$

Am Ende des Netzes F wird eine Summe berechnet (Kostensumme der Komponenten!), dementsprechend liefern wir von der letzten Zelle an alle Vorgänger eine 1.

Fall 0 Der Ausgabeknoten
Wir setzen also:

(152) $l(v_o) := 1$

Nun gehen wir von hinten nach vorne durch. Wichtig ist:in unseren Berechnungsgraphen hat jeder Knoten entweder nur eine Eingabe, oder nur eine Ausgabe, oder beides!

Fall 1 $|\downarrow v| = 1 = |\uparrow v|$.
Nehmen wir einen Knoten v mit $|\downarrow v| = 1 = |\uparrow v|$ Dann setzen wir:

(153) $l(v) := l(v) \cdot l(\downarrow v)$

Wir multiplizieren also die Eingabe von rechts mit dem Wert in $l(v)$. Das funktioniert in allen Fällen gleich!

Fall 2 $|\downarrow v| > 1$
Das bedeutet: wir fügen mehrere Eingaben (von rechts) in v zusammen. Wir setzen dann:

(154) $l(v) := l(v) \sum_{v' \in \downarrow v} l(v')$

Fall 3 $|\uparrow v| > 1$

In diesem Fall ist $l(v) = (1, ..., 1)$. Dann setzen wir einfach:

(155) $l(v) := (l(\downarrow v), ..., l(\downarrow v))$

Nun geben wir an jeden der Vorgänger in einfach den Wert $l(\downarrow v)$ weiter.

Durchlauf Wir erneuern also die Werte von rechts nach links; die Werte der Form $r(v)$ spielen dabei keine Rolle mehr, nur für den forward-step.

Das wird gemacht, bis wir $l(v_i)$ neu gesetzt haben für alle Eingabeknoten v_i.

12.9 Backpropagation: Ausgabe

Wir nehmen an, die Eingabeknoten des Netzes sind $v_{i_1}, ..., v_{i_n}$. Nachdem wir den FF und den BP-Schritt gemacht haben, nehmen wir einfach als Ausgabe des Backpropagation-Algorithmus:

$$\text{Ausgabe von BP: } (l(v_{i_1}), ..., l(v_{i_n}))$$

Was wir nun zeigen müssen ist:

Satz 3 $(l(v_{i_1}), ..., l(v_{i_n})) = \nabla F(\vec{x})$, *wobei* $\vec{x} = (x_1, ..., x_n)$ *unsere Eingabe war.*

Wir haben – was das stimmt – also effektiv den Gradienten ausgerechnet – hurra!

Aber warum ist das so? Hierzu müssen wir uns Fälle anschauen und eine Induktion über das Netz ausführen.

12.10 Beweis von Satz 3

Induktion über die Struktur/Tiefe der Netze.

Induktionsbasis Erstmal nehmen wir an, wir haben einfache Netze ohne hidden-layer. Hiervon gibt es 3 Arten, und es ist nicht schwer zu zeigen dass wir für alle 3 den Gradienten berechnen. Insbesondere gilt das für Netze beliebiger Breite (wegen Addition)! Das bedeutet: wir können für die Induktion uns darauf beschränken, dass wir vorherige Ausgabeknoten als Eingabeknoten zu neuen Knoten auffassen.

Induktionshypothese Nun nehmen wir mal an, wir haben ein Netz N mit

1. Eingabeknoten $v_1, ..., v_n$,
2. Ausgabeknoten $o_1, ..., o_k$.

Wir möchten unser Netz so erweitern, dass es neue Ausgabeknoten $n_1, ..., n_l$ hat, wobei $o_1, ..., o_k$ die Eingaben für diese Knoten sind, also gilt:

$$\text{Für alle } i : 1 \leq i \leq l, \uparrow n_i = \{o_1, ..., o_k\}$$

Wir haben konstruieren also ein dichtes Netz. Am Ende fügen wir weiterhin einen einzelnen Knoten k hinzu, die Werte addiert (die Kostenfunktion).

Nach unserer Annahme haben wir nicht nur ein Netz, sondern eine *Menge von Netzen*, nämlich

- eines mit Knoten $\uparrow^* o_1$,
- ...
- eines mit Knoten $\uparrow^* o_k$

Nach Induktionshypothese berechnen wir für jedes dieser Netze, mit Backward-Eingabe 1, den richtigen Gradienten. Wir konstruieren nun erstmal das neue Netz basierend auf

$$\uparrow^* n_1.$$

Wir haben zunächst die konstante Multiplikation (Kanten), die wie folgt funktioniert:

(156) $\nabla(w \cdot f(x)) = w\nabla f(x))$

(Produktregel für Ableitungen/Gradienten!)
Als nächstes werden die Werte summiert; hier haben wir

(157) $\nabla(f_1(\vec{x}) + ... + f_k(\vec{x})) = \nabla f_1(\vec{x}) + ... + \nabla f_k(\vec{x})$

(Summenregel, für Ableitungen und Gradienten).

Das bedeutet, wir bekommen den Gradienten an dieser Stelle und können die entsprechende Box ausfüllen (Verwaltungsarbeit).

Als nächstes kommt wohl eine nichlineare Funktion g mit Ableitung g'. Wir notieren in dem Knoten also den Wert

(158) $g'(f(\vec{x}),$

$f(\vec{x})$ ist hier der Wert der vorwärts weitergereicht wird.

Nun betrachten wir das neue Netz mit Ausgabe n_1, das berechnet die Funktion f_{n1}. Per Definition haben wir

(159) $f_{n1}(\vec{x}) = g(w_1 f_{o1}(\vec{x}) + ... + w_k f_{ok}(\vec{x}))$

Dementsprechend ist nach Kettenregel

(160) $f'_{n1}(\vec{x}) = g'(w_1 f_{o1}(\vec{x}) + ... + w_k f_{ok}(\vec{x})) \cdot (w_1 f'_{o1}(\vec{x}) + ... + w_k f'_{ok}(\vec{x}))$

Das ist aber genau das, was unser Backprop-Algorithmus berechnet:

1. Er nimmt die Eingabe 1, multipliziert sie mit $g'(w_1 f_{o1}(\vec{x})+...+w_k f_{ok}(\vec{x}))$.
2. Das wird dann im nächsten Schritt multipliziert
   (a) mit $w_1 f'_{o1}(\vec{x})$ an Knoten $o1$,
   (b) ...
   (c) mit $w_k f'_{ok}(\vec{x})$ and Knoten ok.
3. Das dies das vorige layer ist, nehmen wir an dass alle weiteren Vorgänger zu allen Vorgängern Verbunden sind. Das bedeutet: an jedem nächsten Vorgänger haben wir die Summe

$$\begin{aligned} & g'(w_1 f_{o1}(\vec{x}) + ... + w_k f_{ok}(\vec{x})) \cdot w_1 f'_{o1}(\vec{x}) \\ & \qquad +... \\ & + g'(w_1 f_{ok}(\vec{x}) + ... + w_k f_{ok}(\vec{x})) \cdot w_k f'_{ok}(\vec{x}) \\ = g'(w_1 f_{o1}(\vec{x}) + ... + w_k f_{ok}(\vec{x})) & \cdot (w_1 f'_{o1}(\vec{x}) + ... + w_k f'_{ok}(\vec{x})) \end{aligned} \tag{161}$$

   was genau einer Anwendung der Kettenregel entspricht.

Das funktionert aber nur, falls *alle* Zellen, die die neue Zelle aktivieren, dieselbe Menge von Vorgängern haben. Der allgemeinere Fall ist etwas komplizierter, lässt sich aber mit ähnlichen Mitteln abdecken!

12.11 Gradient descent

Wichtig ist es zu verstehen: der BP-Algorithmus verbessert nicht unser Netz, er ist auch erstmal kein Trainingsalgorithmus; er ist nur eine Methode, den Gradienten zu berechnen. Damit können wir aber erstmal nichts anfangen: wir haben ja nur

(162) $\nabla F_{\vec{y}}(\vec{x})$

also den Gradienten für Eingabe $\vec{x}$, berechnet für die Kosten gegeben die Gold-Daten $\vec{y}$. es würde nicht das geringste bringen, nun anstelle von $\vec{x}$ zu setzen

(163) $\vec{x}' := \vec{x} + l\nabla F_{\vec{y}}(\vec{x})$

Wir möchten ja nicht die Eingaben ändern, sondern die Parameter.

Was wir machen können ist aber folgendes: wir stellen das Netz F dergestalt um, dass die **Parameter des ersten layers** die Eingaben werden, die Eingaben die Parameter. Nimm hierfür an, das erste layer besteht aus Knoten $v_1, ..., v_i$, dass zweite layer aus Knoten $v_{i+1}, ..., v_j$; das Gewicht, das zwei Knoten v_n und v_m verbindet, ist w_{mn}. Wir nehmen nun an, es gibt $i \cdot j$ Eingabezellen im ersten layer; nennen wir die n_{nm}, wobei jedesmal der Eingabewert von n_{nm} auf w_{nm} gesetzt wird. Wir verknüpfen den neuen Knoten n_{mn} mit den alten Knoten v_n, wobei das Gewicht der Verbindung x_m beträgt.

Vorsicht: das resultierende Netz ist nicht dicht im intuitiven Sinne dass alle Knoten des ersten layers mit allen des zweiten verknüpft sind, im Gegenteil:

- für zwei Knoten $v_n, v_{n'}$ des zweiten layers sind die Mengen der Vorgänger jeweils disjunkt!

Das passt aber durchaus zu unserer Definition eines dichten Netzes, d.h. der BP-Algorithmus kann auf diesem Netz arbeiten!

GRAFIK!

Auf dieses neue Netz können wir nun BP anwenden, und setzen für den Vektor $\vec{w} = (w_1, ..., w_i)$ der Parameter des ersten layers

(164) $\vec{w}' := \vec{w} + l\nabla F_{\vec{y}}(\vec{w})$

Nun haben wir das erste layer "trainiert" – hurra!

Wir nehmen nun das ursprüngliche Netz F, ersetzen jeden Parameter w durch w'.

Wir geben diesem Netz wiederum die alte Eingabe $\vec{x}$. Das führt an jedem Knoten des zweiten layers zu einer konkreten Eingabe $x_{v_{j+1}}, ..., x_{v_m} \in \mathbb{R}$.

Was wir nun machen ist: wir nehmen diese Ausgaben des ersten layers als Konstanten im zweiten layer; dadurch können wir – immer im Hinblick auf $(\vec{x}, \vec{y})$! – das erste layer *komplett weglassen*. Das resultierende Netz hat nun ein layer weniger.

GRAFIK!

Wir können nun dieselbe Umstellung wie eben machen: die Parameter der zweiten Reihe werden zu Eingaben umgeformt – und nun machen wir dasselbe wie eben.

Dann updaten wir das zweite layer nach demselben Prinzip.

Wir berechnen die neue Ausgabe des zweiten layers bzw. die Eingabe der an die Knoten des dritten layers; dann können wir das zweite layer eliminieren etc. – bis wir fertig sind.

Da wir alle neuen Parameter immer speichern, können wir dann unser komplettes Netz wieder rekonstruieren, mit neuen Parametern.

Was wir gemacht haben: wir haben unser Netz auf *einem* Datenpunkt *einmal* trainiert.

12.12 Sampling issues

Eine Sache die wir noch berücksichtigen müssen ist: falls wir mehrere Datenpunkt haben, macht es durchaus einen Unterschied welche Punkte wir in welcher Reihenfolge nutzen – was wir machen ist ja nicht analytisch. Theoretisch können wir sogar immer wieder nur einen einzigen Punkt benutzen, und es passiert immer wieder etwas neues.

Auch hier gibt es also viele Möglichkeiten, und am besten nutzt man die üblichen Verdächtigen um die passende Stichprobe zu ziehen. Das führt aber sehr weit vom Thema ab und hat mit neuronalen Netzen eigentlich nichts mehr zu tun.

13 Regularisierung

Regularisierung ist eine Änderung eines Lernalgorithmus mit dem Ziel, den Fehler in der Generalisierung zu reduzieren, nicht aber die Fehler auf den Trainingsdaten. Z.B. können wir verlangen (für ein lineares Modells $L(\vec{x}) = M\vec{x} + \vec{a}$) dass der Einfluss von $\vec{a}$ möglichst gering ist. Das macht man, indem man z.B. als Kostenterm die Zahl $\vec{a}^\top \vec{a}$ hinzufügt. (NB: $\vec{a}^\top \vec{a} \in \mathbb{R}$. Das führt dazu, dass die Daten möglichst gut durch die Neigung der Kurve erfasst werden müssen. Regularisierungsterme sind Hyperparameter, deren Wirksamkeit empirisch geprüft werden muss (mit Testdaten).

14 Word embeddings

14.1 Einleitung

Wir kommen nun zu den Problemen, die sich stellen wenn wir neuronale Netze in NLP anwenden. Das erste und grundsätzlichste ist folgendes: Netze arbeiten mit Daten der Form $\mathbb{R}^n$; in NLP haben wir üblicherweise mit Worten/Sätzen zu tun, also mit *strings* oder Worten im technischen Sinne (ein Wort ist eine Kette $w \in \Sigma^*$, wobei auch ein spezielles Symbol für das Leerzeichen umfassen kann.

Was wir also zuerst machen ist: wir betten unsere Daten – also die Objekte, mit denen wir arbeiten wollen – in einen Vektorraum ein. Für uns sind das je nachdem entweder Worte (im linguistischen Sinn oder Buchstaben); man spricht dann von *word embeddings* und *character emeddings*. Wir suchen also eine Funktion

$$vec : \Sigma^* \to \mathbb{R}^n \tag{165}$$

$$vec : \Sigma \to \mathbb{R}^n \tag{166}$$

Diese Abbildung soll natürlich nicht beliebig sein, sondern wichtige Eigenschaften von Worten *kodieren*. Und natürlich wollen wir diese Abbildungen nicht von Hand erstellen, sondern anhand von Daten automatisch induzieren.

Hierfür gibt es verschiedene Modelle; welches sich besonders gut eignet, hängt nicht nur von dem Modell als solches ab, sondern auch von den Aufgaben, die es erfüllen soll.

14.2 Skip-gram models

Das Skip-gram-Modell basiert auf dem Konzept eines Skip-grams. Das ist im Prinzip ein n-gram, in dem ein (zentrales) Wort ausgelassen wird. Die Aufgabe ist nun, das Skip-gram mittels eines Wortes *vorherzusagen*. Formaler, wir möchten den folgenden Term maximieren:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c \& j\neq i} logP(w_{t+j}|w_t) \tag{167}$$

wobei

- c die Größe des Skip-gram Kontextes,
- T die Größe des Lexikons
- w_t das Skip-Wort (in der Mitte)
- Die Wahrscheinlichkeiten P werden einfach vom Korpus geschätzt.

Das heisst: wir schätzen einfach mittels Daten der Form: sei $n = 5$, dann gibt es einen Satz

$$w_1\ w_2\ ...w_{10} \tag{168}$$

dann wird unser Datensatz:

$$(w_1, w_2), (w_1, w_3), (w_2, w_1), (w_2, w_3), (w_2, w_4), (w_3, w_1)...$$

wobei wir natürlich die Zahl der Vorkommen uns merken müssen. Was wir nun suchen ist: eine Vektorrepäsentation der Worte w, w', so dass ein Netz N, gegeben den Eingabevektor $vec(w_t)$ dasjenige liefert, dass am wahrscheinlichsten im Skip-gram mit w' vorkommt.

Wichtig: es handelt sich hierbei um einen **Fake-task**; das ist nicht unsere eigentliche Aufgabe, sondern eine Aufgabe, die wir nur angehen, um damit ein Netz zu trainieren. Nun ist die Frage: wie sehen unsere Wort-Vektoren aus, auf denen wir das Netz N trainieren? Wir nehmen hier die denkbar einfachste Lösung: wir nummerieren unsere Lexikon L; setzen also $L = \{w_1, ..., w_n\}$. Dann nehmen wir

$$vec_1(w_i) = (\overbrace{0, ..., 0}^{i-1 \text{ mal}}, 1, \overbrace{0, ..., 0}^{n-i \text{ mal}}) \tag{169}$$

Man nennt das *one-hot* Vektor; dass sind einfach sog. **Einheitsvektoren** (eine 1, der Rest 0). Wir können mit diesen Vektoren aus den Skip-Grams eine Klassifikationsaufgabe machen:

Gegeben $vec_1(w)$, finde dasjenige $\vec{v}$ so dass

$$\vec{v} = vec_1\left(argmax P_{skip-gram}(w|v)\right) \tag{170}$$

wobei $P_{skip-gram}$ einfach die Wahrscheinlichkeit liefert, dass w im Skipgram von v auftaucht. Das bedeutet: wir möchten ein Netz trainieren, mittels Einheitsvektoren gut die syntaktische Nachbarschaft von Worten vorherzusagen.

Dieses Netz ist genaugenommen ein einfaches **Perzeptron**, also eine Funktion der Form

$$g \circ M \tag{171}$$

wobei M eine Matrix ist, g eine nichtlineare Funktion (die Softmax-Funktion). Wichtig ist:

- $g \circ M(\vec{x}) \in [0, 1]$ soll eine Wahrscheinlichkeit sein, nämlich die Kookurrenzwahrscheinlichkeit.

Das lineare Layer Wir nehmen an, wir haben ein (lineares) Layer M, was unmittelbar nach dem Eingabelayer kommt. Da die Eingabe

$$|Lex| \text{ Dimensionen}$$

hat, ist

$$M \text{ eine } m \times |Lex|\text{-Matrix}$$

, wobei m eine "vernünftige" Zahl ist (wie 100). Was nun passiert ist: $vec_1(w)$ *aktiviert* eine Spalte in M:

$$Mvec_1(w) = M_{-,i} \tag{172}$$

wobei i diejenige Komponente von $vec_1(w)$ ist, die 1 hat. Der *one-hot* Vektor aktiviert also einfach eine Spalte von M – das wars! Mittels dieses ersten Layers haben wir also die Dimension bereits auf m reduziert. Diese Zahl kann man sich vorstellen als

$$m \cong \text{Anzahl der Merkmale}$$

Wir geben also jedem Wort eine Anzahl von Zahlen als "Struktur" mit, und nehmen einfach am Ende diese Gewichte als Repräsentation – soz. Kategorie – des Wortes.

Üblicherweise wird nun keine nichtlineare Funktion appliziert, sondern einfach eine lineare; am Ende aber wird softmax genommen.

Das nichtlineare Layer Die Ausgabewahrscheinlichkeit $P(w|v)$ berechnet man mittels der Softmax Funktion (die unter nichtlinear läuft). Hier repräsentiert M das lineare layer.

$$P_N(w|v) = \frac{exp(Mvec_1(w)^\top Mvec_1(v))}{\sum_{w' \in Lex} exp(Mvec_1(w')^\top Mvec_1(v))} \tag{173}$$

Die Softmax-Funktion garantiert uns, dass wir am Ende eine Wahrscheinlichkeitsverteilung bekommen.

Dieses Netz wird – in diesem Falle – optimiert für M, d.b. für die eine Spalte in M, die durch $vec_1(w)$ aktiviert wird. Wir können nun mit den gewohnten Methoden das neuronale Netz trainieren; allerdings gibt es ein kleines Problem: wir haben oft

$$|Lex| \approx 10^7$$

und die Kosten um

$$\nabla log(P(w|v)) \tag{174}$$

zu berechnen sind proportional zu dieser Größe. Das wiederum ist nur *ein* Paar von

$$|lex|^2 \text{ Paaren von Worten!}$$

D.h. auch wenn wir den Nenner von 173 nur einmal (pro Trainingsrunde) berechnen müssen, müssen wir doch eine Vielzahl von Berechnungen ausführen.

Die Kostenfunktion Mit unserer Skip-Gram Methode können wir natürlich ohne weiteres Wahrscheinlichkeiten schätzen:

$$\hat{P}(w,v) = \frac{\text{Anzahl der Vorkommen } (w,v) \text{ in den Daten}}{\text{Anzahl der Vorkommen } (x,v) \text{ in den Daten (beliebiges } x)} \tag{175}$$

Wir können daraus die Kostenfunktion ableiten:

$$K(w,v) = \frac{1}{2}N(w,v) - \hat{P}(w|v) \tag{176}$$

Hierauf wird das Netz nun optimiert mit den gewöhnlichen Algorithmen. Wichtig ist es, im Kopf zu behalten: das alles hier war nur ein Fake-task: in Wirklichkeit geht es darum, Worte in Vektoren einzubetten!

Die Einbettung Das geht nun wie folgt: sei M' die Matrix unserer fertig optimierten Modells. Wir setzen nun:

$$vec(w) = M'vec_1(w) \tag{177}$$

Anders gesagt: jedes Wort w wird also eingebettet als die *Spalte* der Matrix des Modells, die von $vec_0(w)$ aktiviert wird. Erinnern wir uns dass diese Spalte eine "überschaubare" Größe hat, die von uns selbst festgelegt wird. Wir können damit also *Lex* einbetten in einen Vektorraum mit einer überschaubaren Dimension.

Die zugrundeliegende Annahme, die diese Einbettung rechtfertigt, ist folgende: unsere Vektorrepräsentation ist daraufhin optimieren, Kookkurenzen vorherzusagen, das bedeutet: diese Information (Kookkurrenz) steht irgendwo in diesen Zahlen – denn sonst könnte *Softmax* sie ja nicht herausholen. Wir nehmen aber an, dass jede Einbettung, die die Kookkurrenz-Information kodiert, auch allgemein alle relevanten syntaktischen/semantischen Merkmale kodiert. Wohlgemerkt: das muss nicht so sein, stellt sich aber als empirisch oft zutreffend heraus.

14.3 Hierarchical Softmax

Das Problem im Softmax wie oben ist: er ist furchtbar schwierig zu berechnen, sprengt also in der Praxis oft unsere Möglichkeiten. Es gibt allerdings eine effiziente Art, Softmax zu approximieren, nämlich *hierarchical softmax* (HS). Es reduziert die $|Lex|$ Evaluierungen, die wir vornehmen müssen in *einer* Berechnung der Softmax-Funktion, zu

$$log_2|Lex|$$

Evaluierungen. Hierzu müssen wir uns folgendes vorstellen: Im Prinzip ist unser Modell F (mit Fehlerfunktion!) eine Funktion

(178) $F : \Sigma^* \times \Sigma^* \to [0, 1]$

Wir nehmen also *zwei* Worte, wovon eines als gegeben, das andere als variabel betrachtet wird. Wir können eine solche Funktion auch als einen **Baum** auffassen: wir legen eine erste Eingabe v fest; was fehlt ist die zweite Eingabe w. Wir nehmen nun einen **binären Baum**, wobei:

- jedes Blatt ein Wort $w \in Lex$ repräsentiert; das Blatt mit w liefert uns dabei die Wahrscheinlichkeit $P(w|v)$.
- jeder innere Knoten repräsentiert die Summe der Wahrscheinlichkeiten (also die Wahrscheinlichkeitsmasse) aller unter ihm liegenden Knoten. Das definiert sich also von den Blättern aufwärts induktiv.

Hierbei ist folgendes wichtig:

- Der Baum wird $|Lex|$ Blätter haben, aber:
- jeder Pfad von der Wurzel zu einem Blatt wird $\leq log_2|Lex|$ Knoten umfassen – also *deutlich* weniger.

Damit definiert dieser Baum eine Art *random walk* von der Wurzel zu Worten, wo wir jedesmal mit einer gewissen Wahrscheinlichkeit einen gewissen Abzweig nehmen. Wir bezeichnen nun:

$$n(w, j) \cong \text{der } j\text{-ten Knoten auf dem Pfad von der Wurzel zu } w$$
$$L(w) \cong \text{die Länge des Pfades von der Wurzel zu } w$$

Also gilt: $n(w, 1)$ ist die Wurzel, $n(w, L(w)) = w$. Sei x ein beliebiger innerer Knoten des Baumes. Wir lassen $succ(x)$ einen belieben, aber *eindeutigen* Nachfolger von x denotieren; haben also eine Funktion *succ* die für jeden inneren Knoten einen Nachfolger wählt. Weiterhin sagen wir:

Für eine Gleichung $t = s$ gilt

$$(179)\quad \|t = s\| = \begin{cases} 1, \text{ falls } t = s \text{ wahr ist} \\ -1 \text{ andernfalls .} \end{cases}$$

Außerdem haben wir eine Vektorrepräsentation für jeden inneren Knoten von des Baumes. Da aber die Anzahl der inneren Knoten $\leq$ die Anzahl der

Blätter, fällt das nicht besonders ins Gewicht. HS definiert $P(w|v)$ nun wie folgt:

(180)
$$P(w|v) = \prod_{j=1}^{L(w)-1} S\left(\|n(w,j+1) = succ(n(w,j))\| \cdot Mvec(n(w,j))^\top Mvec(v)\right)$$

wobei S die Sigmoid-Funktion ist. Es lässt sich zeigen, dass das eine konsistente Wahrscheinlichkeitsverteilung erzeugt. Das wichtige hieran ist:

- Die Anzahl der Berechnungsschritte für $(log)p(w|v)$ und $\nabla(log)p(w|v)$ wächst proportional zu $L(w)$;
- Im Allgemeinen gilt $L(w) \approx log_2(|Lex|)$

Aber wie verhält sich Softmax zu hierarchical Softmax?

14.4 Sog. "Additive Kompositionalität"

Ein berühmtes praktisches Beispiel ist folgendes:

(181) $(vec(Madrid) - vec(Spain)) + vec(France) \approx vec(Paris)$

wobei $\approx$ bedeutet: es gibt kein $w \in Lex$ so dass $vec(w)$ näher an $vec(Paris)$ liegt als der Vektor in (181).

Eine wichtige Eigenschaft von Einbettungen ist also folgende: wir möchten semantische Relationen in numerische Übersetzen können semantische Verwandschaft in geometrische Nähe; und wir möchten die Komposition von komplexen Bedeutungen aus einfachen übersetzen in numerische Operationen des Vektorkalküls, z.B. Addition (als einfachste der Operationen der linearen Algebra. Das hieße z.B.:

(182) $vec(kleinesAuto) \approx vec(kleines) + vec(Auto)$

wobei $\approx$ wiederum bedeutet: die beiden Vektoren zielen in ähnliche Regionen, oder, etwas abstrakter, sie bezeichnen einen ähnlichen Winkel.

Um das zu verstehen, muss man folgende Gleichung kennen: wir haben

(183) $\vec{x}^\top \vec{y} = |\vec{x}| \cdot |\vec{y}| \cdot cos(\angle(\vec{x}, \vec{y}))$

wobei $\angle(\vec{x}, \vec{y})$ den Winkel der beiden Vektoren bezeichnet. Wir haben (erinnern wir uns)

$$(184)\quad cos(\angle) = \frac{\text{Ankathete von } \angle}{\text{Hypothenuse}}$$

im rechtwinkligen Dreieck, also:

$$(185)\quad cos(0^\circ) = 1$$

Wir können also sagen: zwei Vektoren $\vec{x}, \vec{y}$ sind sich näher (modulo Länge), je näher $cos(\angle(\vec{x}, \vec{y})$ an 1 liegt. Das lässt sich mittels der obigen Gleichung wiederum leicht ausrechnen (Termumformung). In diesem Fall interessiert also nur die Richtung des Vektors, nicht die Länge.

Das illustriert mithin: wenn wir Worte als Vektoren auffassen, ist nicht völlig klar welchen Merkmalen wir welche Bedeutung zumessen sollen.

15 Recurrent Neural Networks

15.1 Motivation

Ein größer Nachteil von neuronalen Netzen ist, dass die Eingaben eine festgelegt Länge, d.h. Anzahl von Komponenten haben. Oftmals wollen wir aber Sequenzen verarbeiten, die beliebig lang sind! Ein weiteres Problem ist, das wir gerade im NLP unsere Eingaben immer *vorverarbeiten* müssen, damit sie das passende Datenformat $\mathbb{R}^n$ haben. Das können wir gut machen, wenn unser Eingaberaum endlich und von vornherein bekannt ist, wie z.B. das deutsche/englische Lexikon. Aber wie soll das gehen mit einem unbekannten Raum wie den deutschen/englischen Sätzen?

Hier würden wir gerne ein neuronales Modell haben für die Verarbeitung von Sequenzen, im Sinne von beliebigen Sequenzen von Vektoren. Das machen rekurrente neuronale Netze.

15.2 RNN – zwei Auffassungen

Mathematisch gesehen ist ein RNN eine Funktion

$$(\mathbb{R}^n)^+ \to \mathbb{R}^m,$$

wbei n die Eingabedimension des Netzes ist, m die Ausgabedimension. Wir nehmen also eine Folge (beliebiger Länge) von Eingaben, und liefern daraus eine Ausgabe. Da aber das RNN eine Eingabe nach der anderen sequentiell abarbeitet, gibt es dabei auch eine Folge von intermediären Ausgaben, d.h. in diesem Sinne haben wir eine Funktion

$$(\mathbb{R}^n)^+ \to (\mathbb{R}^m)^+$$

Wie wir das Netz auffassen, hängt von der Anwendung ab: falls wir ein Sprachmodell wollen, dann interessiert uns nur die letzte Ausgabe, die wir als Wahrscheinlichkeit auffassen; wenn wir ein Modell zur maschinellen Übersetzung nutzen möchten, dann interessieren uns auch alle intermediären Ausgaben. Wir bezeichnen die beiden Modelle mit RNN RNN^* Wohlgemerkt: das Netz kann in beiden Fällen vollkommen gleich aussehen, wir interpretieren es nur anders.

15.3 Definition

RNNs sind Netze, die nicht einzelne Eingaben verarbeiten, sondern Abfolgen von Eingaben, also Folgen von Vektoren, deren Länge nicht von vornherein festgelegt ist. Der Witz ist hierbei, dass RNNs Netzwerke sind, die sich mit jeder Eingabe ändern, also je nachdem, welche Eingabe(n) sie vorher bekommen haben, eine andere Funktion berechnen. Wenn wir ein normales Netzwerk als eine Funktion N auffassen, dann können wir ein RNN als eine sich mit jeder Eingabe ändernde Funktion auffassen, ähnlich einem Automaten, der mit jeder Eingabe seinen Zustand ändert und dementsprechend eine andere Funktion berechnet.

Entscheidend ist hierbei der **hidden state** des Netzes. Das ist nichts anderes als ein Vektor

$$s \in \mathbb{R}^n$$

Wir kodieren also den Zustand des Netzes als Vektor – nichts weiter. Das Netz am Zeitpunkt $t+1$ nimmt nun zwei Eingaben:

1. Die $t+1$te Eingabe aus der Eingabesequenz, und
2. den Zustandsvektor s_n.

GRAFIK!

Jetzt ist die Frage, wie sich Zustands- und Ausgabevektoren gegenseitig beeinflussen, bzw. wie die Berechnungen ablaufen. Hierzu brauchen wir zwei Funktionen:

1. N nimmt *zwei* Vektoren als Eingabe, nämlich s_i und x_{i+1}, und liefert die Ausgabe s_{i+1} (also den neuen Zustand!)
2. O nimmt s_{+1} (die Zustandsausgabe von N) als Eingabe, und liefert die Ausgabe y_{i+1}

Was hier auffällt ist die "eigentliche" Funktion des Netzes darauf abzielt, den neuen Zustandsvektor zu liefern. Das hängt zusammen mit unserer Auffassung dass die eigentlich interessante Ausgabe die letzte ist. Wir haben folgende Gleichungen:

$$RNN^+(\vec{x}_1, ..., \vec{x}_n, s_0) = (\vec{y}_1, ..., \vec{y}_n) \tag{186}$$

$$RNN(\vec{x}_1, ..., \vec{x}_n, s_0) = \vec{y}_n \tag{187}$$

wobei

$$\vec{y}_i = O(\vec{s}_i) \tag{188}$$

$$\vec{s}_i = N(\vec{s}_{i-1}, \vec{x}_i) \tag{189}$$

wobei gilt: n die Eingabedimension, m die Ausgabedimension, und

$$\vec{x}_i \in \mathbb{R}^n,\ \vec{y}_i \in \mathbb{R}^m,\ \vec{s}_i \in \mathbb{R}^{f(n)}$$

wobei f eine Funktion ist, die durch N bestimmt wird. Wichtig ist:

- ▷ N, O sind die zugrundeliegenden Funktionen, und diese bleiben gleich über alle Berechnungsschritte hinweg.
- ▷ Information über vergangene Berechnungen werden nur weitergegeben durch den Zustandsvektor $\vec{s}$.

Nun gibt es folgende Fragen bzw. Parameter des Netzes:

1. Welche Art von Funktionen sind O, N? Antwort: im einfachen Fall N ein Perzeptron, O eine (nicht)lineare Funktion
2. Wie kommen die Vektoren x, s zusammen in N? Antwort: verschiedene Möglichkeiten, z.B. Addition, oder aber zwei Eingabelayer, also im Prinzip dasselbe wie eine Eingabe $x' = (x, s)$.
3. Was sind die Parameter? Das sind immer dieselben, durch alle Berechnungsschritte.

Als Beispiel für die Funktionsweise können wir die Definition einmal ausschreiben:

$$s_4 = N(s_3, x_4) \tag{190}$$

$$= N(N(s_2, x_3), x_4) \tag{191}$$

$$= N(N(N(s_1, x_2), x_3), x_4) \tag{192}$$

$$= N(N(N(N(s_0, x_1), x_2), x_3), x_4) \tag{193}$$

(194)

Ebenso gilt:

$$y_4 = O(N(s_3, x_4)) \tag{195}$$
$$= O(N(N(s_2, x_3), x_4)) \tag{196}$$
$$= O(N(N(N(s_1, x_2), x_3), x_4)) \tag{197}$$
$$= O(N(N(N(N(s_0, x_1), x_2), x_3), x_4)) \tag{198}$$
(199)

In diesem Sinne enthalten s_n und y_n alle Informationen aller Eingaben. Soweit so gut. Die Frage ist: wie trainieren wir diese Modelle? Denn wir müssen ja nicht nur auf eine Ausgabe optimieren, sondern auch darauf, dass die nächste Eingabe richtig verarbeitet wird!

15.4 Das *bag of words*-Modell CBOW

Ein ganz besonders einfaches RNN-Modell ist das CBOW (*continuous bag of words)*-Modell. In diesem Fall haben wir:

$$s_i = s_{i-1} + x_i \tag{200}$$

$$y_i = s_i \tag{201}$$

wobei

$$s_i, x_i, y_i \in \mathbb{R}^n$$

Wenn wir nun s_0 – den "Startzustand" – entsprechend setzen als

$$s_0 = (0, ..., 0) \tag{202}$$

Dann bekommen wir, für eine Eingabe

$$\vec{x}_1, ..., \vec{x}_i$$

die Ausgabe

$$\vec{y}_m = \sum_{j=1}^{m} \vec{x}_j \tag{203}$$

Wie summieren also, im Falle einer Eingabe

$$w_1 ... w_n$$

einfach die Vektoren

$$vec(w_1) + ... + vec(w_n)$$

Insbesondere ist das Modell also **kommutativ**, d.h. die Reihenfolge der Eingaben spielt keine Rolle.

15.5 SRNN

Wir betrachten nun das Elman Network oder Simple RNN (SRNN). Dieses Modell hat die folgende Form:

$$s_i = N_{SRNN}(x_i, s_{i-1}) = g(Ms_{i-1} + M'x_i + \vec{a}) \tag{204}$$

$$y_i = O_{SRNN}(s_i) = s_i \tag{205}$$

wobei

$$\vec{s_i}, \vec{y_i} \in \mathbb{R}^n,\ \vec{x} \in \mathbb{R}^m,\ M \in \mathbb{R}^{n \times n},\ M' \in \mathbb{R}^{n \times m},\ \vec{a} \in \mathbb{R}^n$$

und g ist eine nichtlineare Funktion, z.B. die Sigmoidfunktion

$$g = \frac{1}{1 + e^{-x}} \tag{206}$$

Das bedeutet: die Ausgabe ist gleich dem Zustandsvektor (hat also keine unabhängige Existenz; der Zustandsvektor wird einfach mit einem Perzeptron berechnet. Die beiden Eingaben werden dabei aufaddiert.

Durch die Präsenz einer nichtlinearen Funktion ist das Modell aber nicht mehr kommutativ, d.h. wir können Information über die Reihenfolge der Eingaben kodieren.

15.6 RNNs stapeln

Bislang haben wir RNNs angeschaut, wo die Berechnung in jedem Zeitschritt im Prinzip ein einfaches Perzeptron war. Wenn wir tiefe RNNs bauen, dann erweitern wir normalerweise *nicht* tiefe Netze, um sie rekurrent zu machen, sondern im Gegenteil:

- ▷ Wir nehmen mehrere einfache RNNs, und **stapeln** RNNs aufeinander.
- ▷ Auf diese Art bekommen wir ein tiefes RNN.

Das bedeutet: wir bauen ein RNN, in dem jedes *Layer* zwei Ausgaben erzeugt:

1. eine Ausgabe die an das nächste Layer weitergereicht wird zur Verarbeitung *derselben* Eingabe
2. eine Ausgabe, die an dasselbe Layer für die Berechnung *der nächsten* Eingabe weitergereicht wird.

Das bedeutet: RNNs werden aufeinander gestapelt, oder anders gesagt: alle internen Layer des Netzes kommunizieren lokal untereinander.

Die **Frage** ist: wie funktioniert diese Veränderung, bzw. wie kommunizieren die Instanzen des Netzwerkes über verschiedene Eingaben hinweg?

Die **Antwort**: jedes einfache RNN kann als ein hidden layer im tiefen RNN funktionieren.

Das bedeutet, wir bekommen in voller Allgemeinheit folgende Definitionen: Sei

- Sei $N_1, ..., N_i$ die Funktionen des ersten,..., $i$ten Layers
- Ebenso $O_1, ..., O_i$

Wir haben also ein RNN der Tiefe i. Nimm nun eine Eingabe

$$\vec{x}_1, ..., \vec{x}_n$$

Dann ist also:

(207) $\vec{y}_1 = O_i \circ N_i(...(O_1 \circ N_1(\vec{x}_1, \vec{s}_0^1))...), \vec{s}_0^i)$

Desweiteren bekommen wir eine Liste von Zuständen

$$\vec{s}_1^{\,1} = N_1(\vec{x}_1, \vec{s}_0^{\,1}) \tag{208}$$

$$\vec{s}_1^{\,2} = N_2(O_1(N_1(\vec{x}_1, \vec{s}_0^{\,1})), \vec{s}_0^{\,2}) \tag{209}$$

$$\vdots \tag{210}$$

$$\vec{s}_1^{\,i} = N_i(...(N_2(O_1(N_1(\vec{x}_1, \vec{s}_0^{\,1})), \vec{s}_0^{\,2})...), \vec{s}_0^{\,i}) \tag{211}$$

Wir berechnen im nächsten Schritt also

$$\vec{y}_2 = O_i \circ N_i(...(O_1 \circ N_1(\vec{x}_2, \vec{s}_1^{\,1}))...), \vec{s}_1^{\,i}) \tag{212}$$

Das bedeutet: im tiefen RNN kommunizieren alle layer "mit sich selber" über Zeitschritte; daher gilt: es ist besser sich ein tiefes RNN als ein "gestapeltes" System von einfachen RNNs vorzustellen, als ein tiefes Feedforward-Netz, das rekurrent wird.

15.7 Training von RNNs

RNNs einfalten/ausfalten Wichtig ist: nehmen wir ein RNN, das eine Eingabe

$$\vec{x}_1, ..., \vec{x}_n$$

verarbeitet. Wir können uns das vorstellen als ein tiefes Feedforward-Netz, in dem immer neue Eingaben kommen. Der eigentliche Punkt ist: bei allen Berechnungen werden **Parameter geteilt**, es gibt also dieselben Parameter in vielen Rechenschritten. Wenn wir also eine Eingabelänge festlegen, dass ist ein RNN nicht mehr eine Verallgemeinerung eines Feedforward-Netzes, sondern ein Spezialfall. Das ist eine Art, ein RNN aufzufassen, und diese Art ist besonders sinnvoll im Training.

Hier ist der entscheidende Punkt: wir können nicht einen isolierten Parameter ändern, sondern wenn wir an einer Stelle eine Änderung vornehmen, müssen wir gleichzeitig an vielen anderen Stellen eine Änderung vornehmen. Das ist genau das Problem.

Das Problem Der Trick im Training von RNNs ist es also, das ganze Netz als ein sehr tiefes Netz zu sehen (jede Eingabe entspricht einem neuen Layer (von Layern). Eingabe ist s_0, und in jedem neuen Layer bildet die Eingabe x_i einen neue Liste von Parametern, zusammen mit den festgelegten Parametern von N.

Gegeben eine Folge $(\vec{x}_1, ..., \vec{x}_n)$ können wir das RNN also soz. ausbreiten, und dann behandeln wie ein normales Netz; die Eingabedaten werden ohnehin als Parameter behandelt. Es gibt hier nur ein Problem:

- Die Berechnung im RNN, für eine gegebene Eingabe, verlangt dass gewisse Parameter gleich sind:
- wir können nicht das erste layer ändern, ohne das t-te, $2t$-te etc. layer im Berechnungsgraphen zu ändern, wobei t die Tiefe des Netzes ist.
- Das geht aber mit unserem bisheriged Backpropagation-Algorithmus nicht!

Backpropagation through time Wir brauchen also einen neuen Algorithmus, nämlich **Backprogation through time** (BPTT).

Der basiert auf folgender Methode: wir berechnen für das ausgefaltete Netz, für Eingabe $(\vec{x}_1, ..., \vec{x}_n)$, die Gradienten **schrittweise von hinten nach vorne**. Wir berechnen also zunächst nur den Gradienten für

$$F(N(\vec{x}_1, ..., \vec{x}_{n-1}), \vec{x}_n) \tag{213}$$

wobei wir natürlich im Hinblick auf die Parameter des letzten Berechnungsschrittes ableiten. Das ist natürlich kein Problem, es handelt sich ja in diesem Fall um ein einfaches Netz (normalerweise einfacher als das FF-Netz, da RNNs oft einfacher gestrickt sind). Uns interessiert also

$$\frac{\partial F(N(\vec{x}_1, ..., \vec{x}_{n-1}), \vec{x}_n)}{N(\vec{x}_1, ..., \vec{x}_{n-1})} \tag{214}$$

anders gesagt: wir berechnen den Gradienten nur den allerletzten Rechenschritt. Das können wir natürlich ohne weiteres machen, das Problem ist nur: wenn wir dann die Parameter ändern würden, würden wir auch alle anderen Rechenschritte verändern, also unsere Eingaben invalidieren!

Die Lösung hierfür ist: wir berechnen den Gradienten für *jeden Zeitschritt*, bekommen also damit

$$\nabla_1 F(\theta_1, ..., \theta_i) \tag{215}$$

$$\vdots \tag{216}$$

$$\nabla_n F(\theta_1, ..., \theta_i) \tag{217}$$

Was wir am Ende machen ist folgendes: wir addieren alle diese Gradienten auf, und setzen die neuen Parameter $(\theta'_1, ..., \theta'_i)$ mittels

$$(\theta'_1, ..., \theta'_i) := (\theta_1, ..., \theta_i) + l \sum_{j=1}^{n} \nabla_j F(\theta_1, ..., \theta_i) \tag{218}$$

In diesem Fall sollte l entsprechend klein gewählt werden, also für eine "normale" Lernrate l' gelten:

$$l = \frac{1}{n} l' \tag{219}$$

um die Summe zu "kompensieren". Das bedeutet: die Summanden legen die Richtung fest, in die optimiert wird.

15.8 Vanishing Gradient

Intuitiv ist das Problem folgendes: der Gradient wird, für eine Eingabe der Länge n, je weiter wir mit unserem Algorithmus in Richtung 1 kommen, desto flacher. Der Einfluss von Änderungen am Zeitschritt 1 (z.B.) auf Berechnungen am Zeitschritt n wird je kleiner, desto größer n.

Das bedeutet am Ende: unsere Optimierung wird nicht erkennen, wie spätere Eingaben für frühere Eingaben relevant sind.

16 Zwei Anwendungsbeispiele für RNN

Wir betrachten hier zwei Anwendungen für RNN, eine mit der Funktion RNN, eine mit der Funktion RNN^+. Das erste werden Sprachmodelle sein, also Wahrscheinlichkeitsverteilungen über Sprachen, das zweite Sequence labelling, ein Problem wie es z.B. im POS-tagging auftritt.

Das Problem ist hier jeweils folgendes: wir bekommen als Ausgabe des Modells immer nur einen Vektor bzw. ein Sequenz von Vektoren; was wir aber möchten ist eine diskrete Ausgabe, d.h. ein Wort, bzw. eine Wahrscheinlichkeitsverteilung. Hier werden wir anschauen, wie man von Vektoren zu Verteilungen bzw. diskreten Ausgaben kommt.

16.1 RNN für *sequence labelling*

Nehmen wir an, wir haben ein Modell RNN, dessen Funktion RNN^+ soll folgendes leisten: eine Funktion

(220) $\ell : Lex^+ \to label^+$

wobei gilt

(221) $|\ell(\overline{x})| = |\overline{x}|$

d.h. unsere Abbildung soll jedes Wort der Eingabe mit einem label versehen. Wie wir die Eingabe verarbeiten ist klar: wir nehmen von jedem $w \in Lex$ einfach die Abbildung $vec(w)$, so dass RNN^+ die Eingabe

$$(vec(w_1), ..., vec(w_n))$$

nimmt. Das Problem ist: die Ausgabe wird ebenfalls eine Folge von Vektoren sein; aber wir können nicht einfach $vec^{-1}(w)$ nehmen, da dieses Urbild höchstwahrscheinlich nicht existiert.

RNN als Verteilung über *label* Wir machen als erstes einen Zwischenschritt und sagen: wir möchten, das wir für jedes Eingabepräfix eine Funktion haben:

(222) $RNN(\vec{x}_1, ..., \vec{x}_j) : label \to [0, 1]$

Also eine **Wahrscheinlichkeitsverteilung** über *label*. Man macht das mittels der *softmax-Funktion*. Als erstes nimmt man an, dass wir eine Funktion haben

(223) $RNN_1 : \mathbb{R}^n \to \mathbb{R}^{|label|}$

Wir möchten also soviele Ausgabekomponenten, wie es label gibt. Diese Funktion kann erstmal beliebig sind. Erinnern wir uns, dass

(224) $softmax(x_1, ..., x_n)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$

softmax liefert eine Wahrscheinlichkeitsverteilung im Sinne, dass für

(225) $softmax : \mathbb{R}^n \to \mathbb{R}^n$

gilt:

$$(226)\ \sum_{i=1}^{n} softmax(\vec{x}) = 1$$

d.h. *softmax* verteilt die Wahrscheinlichkeitsmasse auf seine verschiedenen Komponenten. Was wir nun machen ist nun folgendes: jede Komponente der Ausgabe wird einem Objekt in *label* zugeordnet, so dass wir bekommen: für

$$(227)\ label = \{l_1, ..., l_m\}$$

haben wir

$$(228)\ P_{RNN}(l_j|\vec{x}, ..., \vec{x}_n) = (softmax \circ RNN_1(\vec{x}, ..., \vec{x}_n))_j$$

Wir nehmen also die $j$te Komponente der Ausgabe, um die Wahrscheinlichkeit von l_j zu bekommen.

Von RNN zu RNN^+ Im Prinzip ist damit das Modell klar (bis auf das Training!). Wie müssen nur noch definieren: gegeben die Eingabe

$$\vec{x}_1, ..., \vec{x}_i,$$
$$1 \leq j \leq i,$$

setzen wir

$$(229)\ \hat{j} = \underset{m \in \{1,...,|label|\}}{argmax} (softmax \circ RNN_1(\vec{x}_1, ..., \vec{x}_j))_m$$

Also diejenige Komponente der Funktion, die den höchsten Wert (=Wahrscheinlichkeit) hat, gegeben die Eingaben, die wir bis dahin gelesen haben. Dann setzen wir nur noch:

$$(230)\ RNN(\vec{x}_1, ..., \vec{x}_j) = label_{\hat{j}}$$

und zuletzt

$$(231)\ RNN^+(\vec{x}_1, ..., \vec{x}_i) = RNN(\vec{x}_1)...RNN((\vec{x}_1, ..., \vec{x}_i)$$

Was wir also nur brauchen, als letzten Rechenschritt, ist die Komponente mit maximalem Wert zu finden, was relativ einfach geht.

Die Fehlerfunktion Um dieses Netz zu trainieren brauchen wir noch eine Fehlerfunktion, die uns sagt, wie weit wir jeweils *danebenliegen*. Weiterhin gehen wir davon aus, dass unsere Trainingsdaten die Form haben:

$$D \subseteq Lex^+ \times label^+$$

wir haben also am Ende Paare der Form

$$D = \{(\overline{x}_1, \overline{y}_1), ..., (\overline{x}_m, \overline{y}_m)\}$$

Hier steht $(\overline{x}, \overline{y})$ für zwei gleichlange Folgen $(x_1...x_n, y_1...y_n)$.

Für unser Training können wir die finale Ausgabefunktion nicht brauchen: es gibt ja erstmal kein Mass dafür, wie weit zwei Sequenzen von labels voneinander entfernt sind. Stattdessen nehmen wir die Wahrscheinlichkeitsverteilung

(232) $P_{RNN}(-|\vec{x}, ..., \vec{x}_n) = (softmax \circ RNN_1(\vec{x}, ..., \vec{x}_n)) : label \to [0, 1]$

Wir berechnen die Kosten nun einfach mit der negativen log-Wahrscheinlichkeit. Gegeben eine Paar der Länge n

$$(x_1...x_n, y_1...y_n) \text{ (also } n \text{ Zeitschritte) in } D,$$

gilt zunächst für das Präfix

(233) $K(x_1, y_1) = -log\ P_{RNN}(y_1|vec(x_1))$

Wir nehmen also diejenige Komponente aus

$$softmax(RNN(vec(x_1))),$$

die y_1 entspricht. Das ist eine Wahrscheinlichkeit p; und

$$\text{je kleiner } p \text{ ist, desto größer ist } -log\ p \text{ (0 im Falle } p = 1)$$

Das lässt sich jetzt sehr einfach erweitern:

(234) $$K(x_1, y_1) = -logP_{RNN}(y_1|vec(x_1))$$

(235) $$K(x_1x_2, y_1y_2) = -logP_{RNN}(y_1|vec(x_1)) + (-logP_{RNN}(y_2|vec(x_1), vec(x_2)))$$

(236) $$\vdots$$

(237) $$K(x_1...x_n, y_1...y_n) = -\sum_{j=1}^{n} logP_{RNN}(y_j|vec(x_1), ..., vec(x_j))$$

D.h. wir addieren einfach die lokalen log-Wahrscheinlichkeiten; das ist stochastisch korrekt, da die Summe nur die log-Transformation des Produktes ist, wir also die Produktregel anwenden.

Wir trainieren das Modell also darauf, diese Kosten zu minieren, also die Wahrscheinlichkeit der beobachteten Sequenzen zu maximieren. Im Ende tun wir also nichts anderes, als die **Likelihood** zu maximieren, nur dass das mit einem derart komplexen Modell nicht mehr so einfach geht!

Das Problem der Dimension Das funktioniert sehr gut, wenn wir eine relativ kleine Menge von labels haben, wie z.B. im POS-tagging. Man kann aber auch eine Aufgabe wie maschinelle Übersetzung als Sequence-labelling Problem auffassen; das Problem hierbei ist aber, dass die Ausgabe von RNN *sehr* viele Dimensionen haben wir, nämlich in den Millionen. Das macht die Berechnung der *softmax*-Funktion extrem aufwendig, v.a. im Training, wo sie bei jedem Schritt neu berechnet werden muss. Dieses Problem löst man, indem man z.B. *hierarchical softmax* verwendet.

16.2 RNNs als Sprachmodelle

Ein Sprachmodell ist eine diskrete Wahrscheinlichkeitsfunktion

(238) $P : \Sigma^* \to [0, 1]$

Wahrscheinlichkeit ist klar, diskret heißt einfach soviel wie: jedes $w \in \Sigma^*$ bekommt eine Wahrscheinlichkeit $P(w)$ zugewiesen.

Wie können wir RNNs als Sprachmodelle nutzbar machen? Hier müssen wir zunächst die richtige Perspektive einnehmen: wir nutzen dass RNN, um, gegeben eine Eingabe

$$vec(w_1), vec(w_2), ..., vec(w_{n-1}),$$

eine Wahrscheinlichkeitsverteilung zu induzieren, also:

(239) $RNN \cong P_{RNN}(-|w_1, ..., w_{n-1}) : Lex \to [0, 1]$

soll eine bedingte Verteilung simulieren. Damit gibt uns unser RNN also für jede bisherige Eingabe und jede neue Eingabe eine Wahrscheinlichkeit. Damit haben wir im Prinzip ein Sprachmodell: wir definieren

$$\begin{aligned} P_{RNN}(w_1...w_n) = \\ p(N(vec(w_1), s_0) \cdot p(N(vec(w_2), s_1)) \cdot p(N(vec(w_3), s_2)) \cdot \\ ... \\ \cdot p(N(vec(w_n), s_{n-1})), \end{aligned} \tag{240}$$

p ist hier eine Ausgabefunktion, die sicherstellt, dass die Ausgabe eine Wahrscheinlichkeit ist. D.h. wir multiplizieren einfach die "Wahrscheinlichkeiten". Die Tatsache, dass Worte nicht voneinander unabhängig sind (also Wahrscheinlichkeiten bedingte sind), wird durch die Zustände zum Ausdruck gebracht, die ja Wissen über die Vorgänger kodieren können. NB: hier gibt es auch keine Markov-Annahme, da der Zustand s in keiner Hinsicht beschränkt ist!

Diese Wahrscheinlichkeiten kann man natürlich wieder ausrechnen, genau wie im Sequence labelling. Nehmen wir an, unser Lexikon/Alphabet ist Σ. Wir haben dann also eine Ausgabe

$$\vec{x} \in \mathbb{R}^{\Sigma}$$

Wir nehmen an, dass am Ende ein softmax-layer liegt, so dass der Vektor $\vec{x}$ eine Verteilung über Σ repräsentiert. Dementsprechend muss p nur eine Sache machen:

p pickt die Komponente heraus, die dem Eingabewort w_i entspricht.

Schon haben wir ein Sprachmodell. Es bleibt das **Problem der Dimensionalität** (s.o.), unser Lexikon kann sehr groß sein, und daher werden Berechnungen sehr aufwändig.

Die Kostenfunktion Soweit ist es also sehr einfach, ein RNN als Sprachmodell zu nutzen. Das Problem in diesem konkreten Fall ist: woher kriegen wir die Trainingsdaten? Denn die "echten" Wahrscheinlichkeiten von Sätzen sind uns natürlich unbekannt. Hier brauchen wir eine besondere Fehlerfunktion, nämlich die sog. **Kreuzentropie** (siehe unten).

Was machen wir nun? Wir können für jedes Präfix $w_1...w_{n-1}$ eine einfache Wahrscheinlichkeitsfunktion

$$\hat{P}(w|w_1...w_{n-1}) = \frac{count(w_1...w_{n-1}w)}{count(w_1...w_{n-1})} \tag{241}$$

aus den Daten schätzen (vorausgesetzt, das ganze kommt vor!). Diese Funktion wird z.B. einfach nach *Maximum likelihood* geschätzt, evtl. mit etwas *smoothing*; d.h. wir nehmen mehr oder weniger *relative Häufigkeiten* als Wahrscheinlichkeiten. Unser Ziel ist: RNN soll möglichst akkurat sein in Bezug auf $\hat{P}$, d.h.: wir wollen

$$HK(p \circ N, \hat{P}) \tag{242}$$

minimieren. Dementsprechend ist unsere Kostenfunktion am Datenpunkt $\overline{w}$ (ein Satz!)

$$-\sum_{i=1}^{|\overline{w}|} p \circ N(X_i = w_i) \cdot log(\hat{p}(X_i = w_i)) \tag{243}$$

Hier ist X_i eine Zufallsvariable, die dafür steht, wie dass das i-te Wort im Satz einen gewissen Wert annimmt. Diesen Term versuchen wir zu minimieren!

17 Architekturen mit *gates* – LSTM

17.1 Vorspiel

Das Problem bei RNN ist, das bei *backpropagation through time* der Einfluss, den ein früher Faktor auf eine spätere Berechnung haben kann, stark begrenzt ist, weil es keine Möglichkeit gibt, diese Information *unmittelbar* weiterzugeben, denn alle Parameter spielen in allen Berechnungsschritten die gleiche Rolle. Deswegen führt man sog. *gates* ein, die bestimmte Informationen unmittelbar (d.h. unverändert) weiterleiten über beliebig viele Zwischenschritte.

Man kann das ganze automatentheoretisch auffassen: ein RNN ist wie ein Automat; es hat eine Folge von Eingaben

$$\vec{x}_1, ..., \vec{x}_n$$

und nach jeder Eingabe geht es in einen neuen Zustand $\vec{s}_i$; dementsprechend bekommen wir auch eine Zustandsfolge

$$\vec{s}_0 \xRightarrow{\vec{x}_1} \vec{s}_1 \xRightarrow{\vec{x}_2} ... \xRightarrow{\vec{x}_n} \vec{s}_n$$

Das Problem ist: an jedem Zeitpunkt t_i hat unser Netz nur Zugriff auf den unmittelbar vorhergehenden Zustand und die neue Eingabe. Aber wohlgemerkt: das ist kein Problem für Berechnungen an sich, denn da der Zustandsraum unendlich ist, können wir immer irgenwie Informationen kodieren. Das Problem ergibt sich beim Training, wo gewisse Muster für unsere Augen "unsichtbar" bleiben, insbesondere Abhängigkeiten, die zwischen weit voneinander entfernten Eingaben bestehen. Was wir also suchen ist:

- eine Möglichkeit, am Zeitpunkt t_{i+j} unmittelbar auf Information in s_i zuzugreifen, ohne dass die Zwischenschritte einen Einfluss darauf haben.

Es geht also erstmal und allein um **Speicherzugang**. Einen solchen kontrollierten Speicherzugang kann man modellieren mit dem *Hadamard-Produkt* und einem Vektor

$$\vec{g} \in \{0, 1\}^k,$$

wobei k die Dimension der Zustandsvektoren *und* Eingabevektoren ist. Nimm an, wir haben

$$\vec{x}, \vec{s} \in \mathbb{R}^k$$

Dann haben wir also auch $\vec{g} \in \{0,1\}^k$. Außerdem definieren wir

$$\mathbf{1} = (\overbrace{1, ..., 1}^{k \text{ mal}})$$

Dann können wir ein einfaches **gate** definieren mittels der Definition:

(244) $\vec{s}_{i+1} = \vec{g} \odot \vec{x}_{i+1} + (\mathbf{1} - \vec{g}) \odot \vec{s}_i$

Was geschieht hier? Eigentlich ganz einfach:

- das "gate" $\vec{g}$ nimmt sich diejenigen Komponenten von $\vec{x}_{i+1}$, die relevant sind (in einem zu bestimmenden Sinne);
- die anderen Komponenten sind die entsprechenden Komponenten von $\vec{s}_i$, also des vorigen Zustands (das macht $\mathbf{1} - \vec{g}$).

Das ist im Prinzip eine nur eine "Spielzeugarchitektur", denn wir führen eigentlich keine Berechnungen aus, wir entscheiden nur, welche Komponenten wir über welche Rechenschritte wir weitergeben, und welche wir erneuern; dennoch gibt diese Architektur ein Grundverständnis für was später kommt. Vor allem gibt es zwei Probleme hierbei:

1. Die gates sollen nicht nur Parameter aussuchen, sondern auch aktive Berechnungen ausführen.
2. Wir wollen die exakten gates nicht *apriori* (als Hyperparameter) spezifizieren, sondern als Teil des Trainingsprozesses optimieren. Das setzt aber voraus, dass sie **differenzierbar** sind, was bei $\vec{g} \in \{0,1\}^k$ nicht gegeben ist.

Zunächst lösen wir Problem 2, was deutlich einfacher ist: wir sagen einfach:

$$\text{nimm ein } \vec{g}' \in \mathbb{R}^k \text{ und setze } \vec{g} = S(\vec{g}')$$

wobei S die Sigmoid-Funktion ist. Auf diese Art wissen wir:

1. $\vec{g} \approx \vec{g}'' \in (0,1)^k$, und
2. die resultierende Funktion ist differenzierbar

Unser gate berechnet dann

$$S(\vec{g}) \odot \vec{x}.$$

Hier kommt uns die Eigenschaft der S-Funktion zugute, dass nämlich in fast allen Bereichen der Eingabe der Wert sehr nahe bei 1 oder 0 ist (bis auf einen kleinen, kritischen Bereich). Wir können also $\vec{g}$ als ein **approximatives gate** im obigen Sinn auffassen. Gleichzeitig kann auch dieses gate mit backpropagation differenziert und trainiert werden. Damit haben wir die Grundlage für LSTM (und andere Architekturen), die wir als nächstes präsentieren: diese Modelle basieren darauf, dass wir eine zusätzliche Funktion haben, die entscheidet, welche Information (unmittelbar) weitergeleitet wird, und welche nicht.

17.2 LSTM – Definition

Die *long-short-term-memory* Architektur von Hochreiter& Schmidhuber 1997 addressiert genau dieses Problem. Die Idee ist, dass die Zustandsvektor $\vec{s}$ in zwei Hälften gespalten wird, nämlich

1. den (Langzeit)speicher $\vec{c}$, der Information (fast) unverändert über beliebige Strecken weiterreichen kann, wobei die Weitergabe durch gates (wie oben) kontrolliert und optimiert wird; und
2. den Arbeitsspeicher $\vec{h}$, der in jedem Rechenschritt komplett bearbeitet wird, wie im RNN

Wir definieren:

$$\vec{s}_{j+1} = N_{LSTM}(\vec{s}_j, \vec{x}_j) = (\vec{c}_{j+1}, \vec{h}_{j+1}) \tag{245}$$

wobei

$$\vec{c}_{j+1} = \vec{f} \odot \vec{c}_j + \vec{i} \odot \vec{z} \tag{246}$$

$$\vec{h}_{j+1} = \vec{o} \odot tanh(\vec{c}_{j+1}) \tag{247}$$

$$\vec{i} = S(M^{xi}\vec{x}_{j+1} + M^{hi}\vec{h}_j) \tag{248}$$

$$\vec{f} = S(M^{xf}\vec{x}_{j+1} + M^{hf}\vec{h}_j) \tag{249}$$

$$\vec{o} = S(M^{xo}\vec{x}_{j+1} + M^{ho}\vec{h}_j) \tag{250}$$

$$\vec{z} = tanh(M^{xz}\vec{x}_{j+1} + M^{hz}\vec{h}_j) \tag{251}$$

Außerdem wird die Ausgabe definiert als

$$\vec{y}_{j+1} = O_{LSTM}(\vec{s}_j) = \vec{h}_j \tag{252}$$

das bedeutet: der Arbeitsspeicher funktioniert als Ausgabe. Unser Zustand ist also erstmal zweigeteilt; gleichzeitig gibt es 3 gates, nämlich

1. **i**nput,
2. **f**orget
3. **o**utput

Alle drei gates berechnen ein Perzeptron, also eine lineare Funktion + die Sigmoid Funktion, jeweils mit Eingabe $\vec{x}_{j+1}$ und $\vec{h}_j$, also neue Eingabe und alter Arbeitsspeicher.

1. input kontrolliert, wieviel vom neuen Kandidaten $\vec{z}$ in den Speicher einfließt, durch $\vec{i} \odot \vec{z}$

2. forget kontrolliert, wieviel von der alten Speicherbelegung behalten wird, durch $\vec{f} \odot \vec{c_j}$

3. output kontrolliert, welcher Teil des Langzeitspeichers (mit $tanh$) auf den Arbeitspeicher kommt ($\vec{o} \odot tanh(\vec{c_j})$); gleichzeitig ist das ja der Output des Zeitschrittes.

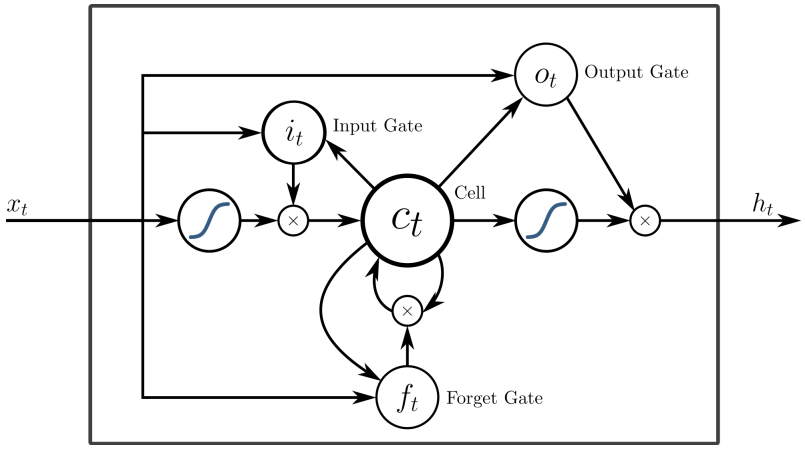
$\vec{z}$ ist ein Kandidat, den (Langzeit)speicher zu aktualisieren; hier wird die $tanh$-Funktion verwendet. Das macht den Unterschied zu den gates: die gates sollen Werte in $[0, 1]$ annehmen (s.o.), während der eigentliche Arbeitsspeicher-Zustand $\vec{h}$ und der Update-Kandidate $\vec{z}$ in $[-1, 1]$ liegen. Wichtig ist auch folgendes: der Langzeit-Speicher $\vec{c}$ ist der *einzige* Vektor in der Architektur, dessen Komponente beliebige Werte in $\mathbb{R}$ annehmen können: hier werden Werte nicht normalisiert (mittels S oder $tanh$). Das bedeutet, wir können uns wirklich Sachen merken, z.B. können wir mit (246) in einer Komponente zählen, wie oft wir einen bestimmten Buchstaben gelesen haben (wir addieren jedesmal ≈ 1 hinzu über $\vec{i} \odot \vec{z}$). In diesem Sinne ist es wirklich ein Langzeitspeicher, auf dem wir Informationen ablegen können. (Als Beispiel: nimm an, für jeden Buchstaben ist eine Komponente zuständig, und die Zahl der Komponente sagt und, wie oft wir den Buchstaben gesehen haben).

Wichtig ist es , zu verstehen was die Parameter des Netzes sind:

- Die Matrizen M^{xi}, M^{hi}, M^{xf}, M^{hf}, M^{xo}, M^{ho} sind die Parameter des Modells, die optimiert werden. Alles andere sind Hyperparameter bzw. Objekte, die erst berechnet werden.

Diese Matrizen sind übrigens alle quadratisch, da alle Vektoren, die hier involviert sind, die gleiche Dimension haben müssen, sonst sind die Operationen $\odot$, $+$ nicht mehr definiert. Die Dimension ist bestimmt durch die Dimension der Eingabevektoren.

Das löst das größte Problem von RNNs, denn: wir können die Parameter so setzen, dass $\vec{c}$ (bzw. eine Komponente davon) durch eine beliebige Anzahl von Schritten hindurch praktisch unverändert bleibt.



LSTM liefern derzeit state-of-the-art Ergebnisse für die meisten Aufgaben, die das Modellieren von Sequenzen erfordern. Ein Problem dieser Architektur ist, das sie relativ kompliziert ist, und daher

1. die Ergebnisse schwer nachvollziehbar sind, und
2. das Training recht komplex.

Eine etwas einfachere Alternative liefern *gated recurrent units* (GRUs).

17.3 Gated recurrent units

GRU basieren auf denselben Ideen von gates, haben aber eine einfachere Architektur und keine separaten Speicherabteile. Sie sind wie folgt definiert:

$$\vec{h}_{j+1} = N_{GRU}(\vec{h}_j, \vec{x}_{j+1}) = (\mathbf{1} - \vec{z}) \odot \vec{h}_j + \vec{z} \odot \tilde{h}_{j+1} \tag{253}$$

wobei

$$\vec{z} = S(M^{xz}\vec{x}_{j+1} + M^{hz}\vec{h}_j) \tag{254}$$

$$\vec{r} = S(M^{xr}\vec{x}_{j+1} + M^{hr}\vec{h}_j) \tag{255}$$

$$\tilde{h}_{j+1} = tanh(M^{xh}\vec{x}_{j+1} + M^{hh}(\vec{h}_j \odot \vec{r})) \tag{256}$$

(257)

Der output ist gegeben durch

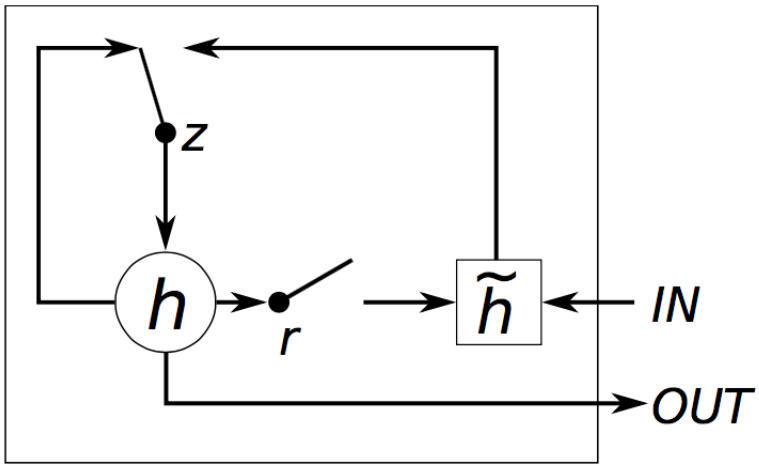
$$y_{j+1} = O_{GRU}(\vec{h}_{j+1}) = \vec{h}_{j+1} \tag{258}$$

In diesem Fall gibt es nur einen Zustand, der aber durch mehrere gates bestimmt wird:

- $\vec{z}$ wird berechnet über ein Perzeptron auf Eingabe+ voriger Zustand; dient als gate für $\vec{h}_j$ und $\vec{z}$
- $\vec{r}$ wird berechnet über ein Perzeptron auf Eingabe+ voriger Zustand; ist ein Hilfsvektor um $\tilde{h}$ zu berechnen
- $\tilde{h}$ ist der "neue Arbeitsspeicher, für den geprüft wird, wie weit er in den Speicher einfließt.

Wiederum ist es wichtig zu verstehen, was die Parameter des Modelles sind:

- Die Matrizen M^{xz}, M^{hz}, M^{xr}, M^{hr}, M^{xh}, M^{hh} sind die Parameter des Modells, die optimiert werden. Alles andere sind Hyperparameter bzw. Objekte, die erst berechnet werden.



18 Convolutional Neural Networks

18.1 Convolution – was ist das?

In einem Satz: CNN sind neuronale Netze, die Konvolution anstelle von Matrix-Multiplikation nutzen in mindestens einem layer. Aber was ist das? Im Prinzip bedeutet Convolution so viel wie: wir beschränken die Freiheitsgrade der Funktion auf eine manuelle, informierte Art und Weise. Das besteht darin dass wir verschiedene Funktionen separat berechnen auf verschiedenen Teilen der Eingabe, und diese dann wiederum auf eine bestimmte Art und Weise zusammenfügen.

In Beispiel 2 werden wir sehen, dass das bedeuten kann dass wir, anstelle eines ganzen Satzes, nur die n-gramme eines Satze betrachten. Informationen werden aus diesen n-grammen extrahiert, und am Ende wieder zusammengefügt.

In Beispiel 1 sehen wir, dass wir für zwei Eingaben zwei separate, relativ einfache Funktionen definieren können, die Berechnungen ausführen; diese beiden Funktionen werden dann zusammengeführt um eine deutlich komplexere Funktion auszuführen. Dadurch dass wir aber sehr sinnvolle Beschränkungen auf den separaten Funktionen haben, wird das Modell leichter zu handhaben.

18.2 Convolution – Beispiel 1

Nehmen wir ein etwas blödes Beispiel aus der Literatur: wir haben einen **Laser-Sensor**, der uns sagen soll, wo ein **Raumschiff** ist; wir möchten also damit die Position bestimmen. Leider ist das Gerät fehleranfällig, von daher ist es sicherer, wenn wir über eine Reihe von Messungen mitteln. Allerdings kann es gut sein, dass das Raumschiff in der Zwischenzeit seine Position ändert. Wir haben also folgende Information:

- Unser Sensor liefert uns wechselnde Information über die Lokalität des Raumschiffs –
- wir wissen aber nicht, ob und zu welchem Grad diese Änderungen durch Messfehler oder Positionswechsel verursacht werden.

Was wir suchen ist die **aktuelle** Position des Raumschiffs. Aus obigen Gründen sollten wir nicht nur die aktuelle Messung berücksichtigen. Allerdings sollten wir die aktuelle Messung **stärker** berüchsichtigen als vergängene Messungen, anders gesagt: es gibt eine Funktion

$$w : \mathbb{R} \to [0, 1] \tag{259}$$

die *monoton fällt*, die uns sagt wie wir unsere Messung berücksichtigen sollen. Wir sagen dann, unser Sensor liefert eine Funktion

$$s : \mathbb{R} \to \mathbb{R}^n \tag{260}$$

wobei das Argument der Zeitpunkt der Messung ist, und der Wert das Messergebnis. Wenn nun ein Messergebnis an a gemessen wurde, dann wird es gewichtet (am Zeitpunkt t) mit

$$w(t - a) \tag{261}$$

Wie implementiert man das? Das hängt nun an der Frage, ob unsere Messungen eine stetige Funktion sind, oder eine diskrete Funktion, also am Ende hängt es an unserem Begriff von Zeit. Im stetigen Fall definieren wir unsere Messfunktion

$$\mu(t) = \int s(a)w(t - a)da \tag{262}$$

wobei a das Alter der Messung ist, $s(t - a)$ der Sensorinput in Bezug darauf. Wir integrieren also über das Alter der Messung. In diesem Fall muss w

eine Wahrscheinlichkeitsdichtefunktion sein, die darüberhinaus für negative Werte 0 liefert, also:

$$\int_{-\infty}^{\infty} w(x)dx = 1 \tag{263}$$

$$x < 0 \implies w(x) = 0 : \tag{264}$$

Wenn w für negative Werte ein positives Gewicht liefern würde, dann würden es uns (konzeptuell gesprochen) erlauben, in die Zukunft zu blicken.

Das unbestimmte Integral liefert nur die stetige Generalisierung der diskreten Funktion; wenn wir unsere Messungen in diskreten Abständen vornehmen (z.B. Sekunden), dann sieht das ganze wie folgt aus:

$$\mu(t) = \sum_{a=-\infty}^{\infty} s(a)w(t-a) \tag{265}$$

Hier wird das Integral also eine (unendliche) Summe. Und genau diese beiden Operationen nennen wir **Konvolution**. Es geht hierbei also um eine **Gewichtung**; man benutzt hier das Symbol * und schreibt also:

$$\mu(t) = (s * w)(t) \tag{266}$$

Man nennt

- die Funktion s den **input**, also die Eingabefunktion;
- die Funktion w nennt man den **kernel**.

In diesem Fall haben wir die Konvolution im Hinblick auf eine Dimension, nämlich die Zeit. Der Begriff ist aber allgemeiner: wann immer wir **strukturierte Daten** haben (durch Zeit, oder Position in einer Fläche, Koordinate in einem Raum etc.) können wir dieses Prinzip anwenden. Normalerweise haben dann aber input und kernel dieselbe Dimension:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{267}$$

Das erste wichtige Ergebnis ist, dass Konvolution **kommutativ** ist:

$$(I * K)(i,j) = (K * I)(i,j) \tag{268}$$

was explizit bedeutet:

$$(269)\quad \sum_m \sum_n I(m,n)K(i-m,j-n) = \sum_m \sum_n I(i-m,j-n)K(m,n)$$

In der Praxis benutzt man oft eine etwas andere Funktion, die man **cross-correlation** nennt, und die wie folgt definiert ist:

$$(270)\quad S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m,j+n)K(m,n)$$

Das ist offensichtlich grob äquivalent (wenn man von Details absieht) zur Konvolution, wird deswegen oft einfach an ihrer Stelle verwendet.

Der wesentliche Punkt ist: wir können μ als eine Funktion auffassen, die wir trainieren mittels unserer Messungen. Wir wissen zwar nie wirklich, wo sich das Raumschiff befindet (haben also keine "Gold-Daten"), aber wir können durchaus vorhersagen machen:

- Ein schnell fallendes w bedeutet: wir nehmen an, dass das Raumschiff sich stark bewegt. Dementsprechend sollte sich der Mittelwert von Messungen relativ schnell verschieben.
- Ein relativ langsam fallendes w bedeutet: das Raumschiff ist relativ statisch, der Mittelwert der Daten bleibt also über verschiedene Zeitfenster relativ gleich.

Damit können wir bereits die Funktion μ trainieren, indem wir verschiedene Vorhersagen machen:

Vorhersage 1 Die folgende Messung wird relativ nah an der letzten Messung sein.

Vorhersage 2 Die folgende Messung wird relativ nah am Mittelwert der bisherigen Messung sein.

Das sind natürlich zwei Extreme, die beste Vorhersage wird irgendwo zwischen den beiden Polen sein. Dennoch wird das Prinzip klar: dadurch dass unsere Funktion μ nur *einen* Parameter hat (w), können wir sie effektiv trainieren obwohl unsere Daten sehr unzuverlässig sind. Das klappt also nur, weil wir μ als Konvolution auffassen – im allgemeinen Fall wären wir verloren.

18.3 Convolution – Beispiel 2

Nehmen wir nun folgendes Beispiel: wir möchten anhand eines Satzes herausfinden, ob er eine positive oder negative Aussage macht (das ist z.B. wichtig bei Kritiken, die es online ja in großer Zahl gibt).

(271) `Das Stück war lebhaft, aber nicht überzeugend.`

Hier sehen wir, dass es nicht genug ist, einzelne Worte zu betrachten – eine Betrachtung des Satzes als ganzes kann aber durchaus zuviel sein. Unser Modell basiert dagegen auf n-grammen, soll also jedes n-gram evaluieren (wir gehen davon aus, dass die Abhängigkeiten hinreichend lokal sind). Wir haben also erstmal, vor allem anderen,

Schritt 1: Word embedding Das haben wir besprochen; wir nehmen an, das

$$vec : Lex \to \mathbb{R}^L$$

das word-embedding ist, $L \in \mathbb{N}$ eine beliebige Konstante, die wir uns merken.

Schritt 2: Die Eingaben Unsere Eingabe ist nun eine Sequenz von Vektoren

$$(\#)\ \vec{x}_1, ..., \vec{x}_n$$

Unsere n-gram Größe legen wir fest auf k. Das bedeutet: wir definieren eine Operation auf

$$(\mathbb{R}^L)^k$$

Diese Operation wird ausgeführt der Reihe nach auf Vektoren

- $\vec{x}_1, ..., \vec{x}_k$
- $\vec{x}_2, ..., \vec{x}_{k+1}$
- ...
- $\vec{x}_{(n-k)+1}, ..., \vec{x}_n$

Wir operieren also der Reihe nach auf diesen Vektoren (von Vektoren).

Schritt 3: Die Convolution Wir definieren nun die Operation, die wir auf (#) ausführen. Zunächst definieren wir die Operation $\bigoplus$ mittels

$$\bigoplus(\vec{x}_1, ..., \vec{x}_k) = (x_1^1, ..., x_1^L, ..., x_k^1, ..., x_k^L) \tag{272}$$

wir schreiben also, in Worten, die Komponenten aller Vektoren in einen neuen Vektor der Grösse

$$\bigoplus(\vec{x}_1, ..., \vec{x}_k) \in \mathbb{R}^{L \cdot k}$$

Einfachheit halber setzen wir

$$\vec{y}_i = \bigoplus(\vec{x}_i, ..., \vec{x}_{i+k}) \tag{273}$$

Dieser Vektor enthält, in einem einfachen Sinne, die Information des n-grams von Worten, und die Operation $\bigoplus$ brauchen wir nur, um möglichst einfach darauf zugreifen zu können. Nun wenden wir auf diese Vektoren ein einfaches Perzeptron an, also eine Funktion

$$g \circ M : \mathbb{R}^{L \cdot k} \to \mathbb{R}^i \tag{274}$$

Das bedeutet:

- $M \in \mathbb{R}^{j \times L \cdot k}$ ist eine Matrix;
- g ist eine nicht-lineare Funktion;
- dazu kommt noch ein **bias**-Term,

so dass wir einen neuen Vektor bekommen. Um das ganze übersichtlich zu machen schreiben wir

$$\vec{p}_i = g(M\vec{y}_i + \vec{a}) \tag{275}$$

wobei natürlich $\vec{a} \in \mathbb{R}^j$. Hier gilt: $\vec{p}_i$ soll die Information des i-ten k-grams kodieren und kann entsprechend trainiert und evaluiert werden. Hier gilt wieder:

▷ Jede Dimension von $\vec{p}_i$ kann eine unterschiedliche relevante Information des n-grams kodieren.

- ▷ Für die n-grame selber spielt die Reihenfolge der Worte durchaus eine Rolle;
- ▷ Die Reihenfolge der n-grame untereinander spielt allerdings keine Rolle (bzw. höchstens als technisches Detail)

Das bedeutet als Zusammenfassung: die Konvolution erlaubt uns hier, aus der Gesamtinformation des Satzes gewisse Information zu behalten, während andere Information als irrelevant weggelassen wird. Wir können das also explizit im Modell (als Hyperparameter) spezifizieren – dann muss unser Modell das nicht mehr lernen, kann sich also auf andere Dinge konzentrieren.

Schritt 4: Pooling Was uns Convolution liefert ist ein Vektor von Vektoren, also nicht die Datenstruktur, die wir im neuronalen Netz weiterverarbeiten können: da brauchen wir einen einfachen Vektor. Pooling heißt also: wir nehmen eine Abbildung

$$pool : (\mathbb{R}^j)^n \to \mathbb{R}^i$$

Hierbei ist wichtig: normalerweise möchten wir die Dimension etwas reduzieren. Die häufigste Methode, um das zu machen, ist das sog. *max pooling*, das (wenn wir die obigen Indizes weiterführen) einen Vektor der Länge i gibt, also:

$$mp : (\mathbb{R}^j)^n \to \mathbb{R}^j \tag{276}$$

was definiert ist durch die max-Operation:

$$(mp(\vec{y}_1, ..., \vec{y}_n))_m = \max_{1 \leq i \leq k}(y_1^m, ..., y_n^m) \tag{277}$$

Das bedeutet: die jeweils $m$te Komponente des i-dimensionalen Vektors ist das Maximum alle $j$ten Komponenten der Eingabevektoren. Intuitiv heißt das:

- Im Convolution-Schritt spezialisiert sich jede Komponente als ein bestimmter Prädiktor, und
- im Pooling-Schritt nehmen wir aus allen Vektoren den Prädiktor heraus, der am eindeutigsten ist.

Ein anderes Modell wäre *average pooling*, was einfach den Durchschnitt statt des Maximums nimmt:

$$(278)\quad (ap(\vec{y}_1, ..., \vec{y}_n))_m = \frac{1}{n}\sum_{l=1}^{n}(\vec{y}_l)_m$$

Eine weitere Alternative ist *k-max-pooling*. Das ist ähnlich wie *max-pooling*, nur dass wir auf jeder Komponente nicht das Maximum suchen, sondern die k-maximalen Werte. Auf diese Art und Weise wird aus einer $n \times i$-Matrix eine $k \times i$-Matrix, wobei $k \leq n$. Z.B.

$$(279)\quad \begin{pmatrix} 1 & 5 & 8 \\ 6 & 9 & 0 \\ 4 & 6 & 1 \\ 9 & 2 & 4 \\ 1 & 7 & 3 \\ 0 & 3 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 9 & 8 \\ 6 & 7 & 7 \end{pmatrix}$$

wenn $k = 2$. Nun haben wir allerdings noch keinen Vektor, sondern eine Matrix. Wir machen daraus einen Vektor, inde wir die Vektor-komponenten konkatenieren:

$$(280)\quad \begin{pmatrix} 9 & 9 & 8 \\ 6 & 7 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 9 & 8 & 6 & 7 & 7 \end{pmatrix}$$

Eine andere Art von Pooling ist *dynamic pooling*. In diesem Fall werden die Vektoren zunächst in Gruppen aufgeteilt. Für jede Gruppe wird z.B. *max-pooling* appliziert; dann haben wir wieder eine Matrix (Liste von Vektoren), die wie im vorigen Fall auf einen Vektor reduziert wird. Dynamic pooling ist allerdings nicht ganz so interessant im allgemeinen Fall, da die Auswahl der Gruppen domänenspezifisches Wissen voraussetzt.

Motivation Aus formaler Sicht ist Convolution eigentlich etwas ähnliches wie *Parameter sharing*, wie wir das bereits bei RNN festgestellt haben (das ausgefaltete Netz berechnet ein feedforward-Netz mit geteilten Parametern). Wenn wir den Eingabesatz wieder als eine einzigen Eingabevektor der Länge $L \cdot n$ sehen, dann bedeutet die Konvolution, dass wir für jedes Fenster der Länge k dieselben Parameter anwenden. Man kann sich das konkret vorstellen als

- ein dünn verknüpftes layer (wir also eben nicht alle Zellen eines layers mit allen des nächsten verbinden)
- in dem viele Kanten dieselben Parameter haben.

ein

19 Ein Ausflug: Entropie, Bedingte Entropie, Kreuzentropie

19.1 Definition

Im maschinellen Lernen geht es oft darum, Wahrscheinlichkeitsverteilungen zu induzieren. In diesem Fall brauchen wir zwei Dinge, nämlich erstens gewisse Referenzverteilung (zumindest auf einem Fragment des Raumes), und zweitens eine Kostenfunktion die uns besagt, wie weit wir davon abweichen. Den ersteren Punkt besprechen wir hier nicht, die Kostenfunktion ist dagegen oft einfach die Kreuzentropie der beiden Verteilungen. Dieses Konzept und das der Entropie besprechen wir hier kurz.

Das Konzept der Entropie formalisiert die *Unsicherheit* in einem System. Die Definition ist wie folgt: wir haben eine Wahrscheinlichkeitsfunktion P und ein Ereignis ω. Die Entropy von ω (nach P), geschrieben $H_P(\omega)$, ist

$$H_P(\omega) := P(\omega) \cdot -log(P(\omega)) \tag{281}$$

Die Entropie eines einzelnen Ereignisses ist normalerweise weniger interessant als die Entropie einer ganzen Verteilung P (über einen diskreten Raum Ω, geschrieben $H(P)$:

$$H(P) := -\sum_{\omega \in \Omega} P(\omega) log(P(\omega)) \tag{282}$$

Es ist leicht zu sehen dass das einfach die Summe der Entropie der Ereignisse ist; wir haben nur das minus ausgeklammert. Als Faustregel lässt sich sagen: in einem Raum mit n Ergebnissen ist die Entropie *maximal*, wenn alle Ereignisse die gleiche Wahrscheinlichkeit $1/n$ haben; sie wird minimal (geht gegen 0), falls es ein Ereignis gibt dessen Wahrscheinlichkeit gegen 1 geht. Das deckt sich mit unseren Intuitionen: je größer die Entropie, desto weniger Sicherheit haben wir, wie das Ergebnis sein wird. Z.B.: nehmen wir das Beispiel eines fairen Würfels; wir können die Entropie des zugehörigen Wahrscheinlichkeitsraumes wie folgt ausrechnen:

```
> x = 0 : 5
> for(i in 1 : 6){
```

```
+x[i] < −1/6 ∗ log(1/6)}
> sum(x)
[1] − 2.584963
```

(Wir verzichten darauf, die Entropie ins positive zu wenden). Wenn wir hingegen annehmen, 5 Seiten haben die Wahrscheinlichkeiten 1/10 und die 6 hat eine Wahrscheinlichkeit 1/2, dann bekommen wir:

```
> x = 0 : 5
> for(i in 1 : 5){
+x[i] < −1/10 ∗ log(1/10)}
> x[6] = 1/2 ∗ log(1/2)
> sum(x)
[1] − 2.160964
```

Andersrum gesagt: je größer die Entropie (einer Wahrscheinlichkeitsverteilung für ein Zufallsexperiment), desto größer der Informationsgewinn, der darin besteht das Ergebnis zu erfahren. Wichtig ist: Entropie ist immer unabhängig von den einzelnen Ergebnissen, es spielt also keine Rolle ob die 1 oder die 6 eine erhöhte Wahrscheinlichkeit hat. Alles was zählt ist eben die Ungewissheit; wir können das mit einem weiteren Versuch nachrechnen:

```
> x = 0 : 5
> for(iin1 : 3){
+x[i] < −1/10 ∗ log(1/10)}
> x[4] = (1/20) ∗ log(1/20)
> x[5] = (3/20) ∗ log(3/20)
> x[6] = 1/2 ∗ log(1/2)
> sum(x)
[1] − 1.777507
```

Die Entropie ist also weiter gesunken, denn wir haben die Wahrscheinlichkeiten weiter ungleich aufgeteilt zwischen 2 Ergebnissen: während also die Entropie für 1,2,3,6 gleich geblieben ist, ist sie für 4,5 lokal gesunken, also ist sie auch global gesunken. Man kann auch umgekehrt sagen: da die uniforme Wahrscheinlichkeitsverteilung für uns den *Mangel* an relevanter Information bezeichnet, gibt es die Korrelation

maximale Entropie $\approx$ maximale Unwissenheit

Darauf basiert eine wichtige Methode der Wahrscheinlichkeitstheorie, die sog. Maximum Entropie Schätzung. Die basiert auf dem Grundsatz:

> In Ermangelung sicherer Information ist es besser, möglichst wenig Sicherheit anzunehmen, als falsche Sicherheit die es nicht gibt (es ist besser zu wissen dass man etwas nicht weiß)

Das bedeutet effektiv: wir sollten die Wahrscheinlichkeitsverteilung annehmen, die

1. mit unserem Wissen kompatibel ist,
2. ansonsten aber die Entropie maximiert.

Man definiert die Entropie auch oft für Zufallsvariablen:

$$H(X) := -\sum_{x \in X} P(X = x) log(P(X = x)) \tag{283}$$

19.2 Bedingte Entropie

Die bedingte Entropie von zwei Variablen (über demselben Wahrscheinlichkeitsraum) ist wie folgt definiert (hier bedeutet $y \in Y$ soviel wie: y ist ein Wert, den Y annehmen kann):

$$H(X|Y) = \sum_{y \in Y} P(Y = y) H(X|Y = y) \tag{284}$$

Wenn wir diese Definition auflösen, bekommen wir:

$$H(X|Y) = \sum_{x \in X, y \in Y} P(X^{-1}(x) \cap Y^{-1}(y)) log \left(\frac{P(X^{-1}(x) \cap Y^{-1}(y))}{P(Y^{-1}(y))} \right) \tag{285}$$

Die bedingte Entropie ist also ein Maß dafür, wie stark die Werte einer Zufallsvariable Y die Werte einer Zufallsvariable X festlegen. Wenn der Wert von X durch den Wert von Y – egal wie er ist – immer festgelegt ist, dann ist

$$H(X|Y) = 0 \tag{286}$$

insbesondere also:

(287) $H(X|X) = 0$

Umgekehrt, falls der Wert von Y keinerlei Einfluss hat auf die Wahrscheinlichkeitsverteilung des Wertes von X, dann haben wir

(288) $H(X|Y) = H(X)$

Es ist klar dass das hier nur für diskrete Wahrscheinlichkeitsräume funktionieren kann; in kontinuierlichen Räumen funktionieren diese Dinge etwas anders.

Es gibt auch eine Kettenregel für bedingte Entropie:

(289) $H(X|Y) = H(\langle X, Y\rangle) - H(Y)$

wobei $\langle X, Y\rangle$ eine neue Variable ist, mit

(290) $P(\langle X, Y\rangle = (x, y)) = P(X = x, Y = y) = P(X^{-1}(x) \cap Y^{-1}(y))$

Wir nehmen also die Entropie der **Verbundverteilung**, und ziehen die Entropie von $H(Y)$ ab.

19.3 Kullback-Leibler-Divergenz

Die KL-Divergenz ist eine andere Art zu messen, wie ähnlich sich zwei Wahrscheinlichkeitsverteilungen P und Q sind. Die Definition ist wie folgt:

(291) $D_{KL}(P\|Q) = \sum_{\omega\in\Omega} P(\omega) log \frac{P(\omega)}{Q(\omega)}$

An dieser Definition kann man ablesen:

1. $D_{KL}(P\|Q) = 0$ gdw. für alle $\omega \in \Omega$ gilt: $P(\omega) = Q(\omega)$; denn $log(1) = 0$.
2. In allen anderen Fällen ist $D_{KL}(P\|Q) > 0$ (das ist nicht wirklich leicht zu sehen).
3. $D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$, d.h. wir haben ein asymmetrisches Maß.

4. Man kann es jedoch symmetrisch machen auf folgende Art und Weise:

$$D_2(P\|Q) = D_{KL}(P\|Q) + D_{KL}(Q\|P) = D_2(Q\|P)$$

Sie gibt uns also ein Maß dafür, wie weit Q von P **entfernt** ist. Dadurch unterscheidet sie sich konzeptuell von $H(X|Y)$, dass bestimmt wie stark X von Y *determiniert* wird.

Man bezeichnet $D_{KL}(P\|Q)$ auch als den **Informationsgewinn**, den man mit P gegenüber Q erzielt. Wenn wir z.B. das obige Kodierungsbeispiel fortführen, dann sagt uns $D_{KL}(P\|Q)$, wieviel Platz wir (im Durchschnitt) verschwenden, wenn wir eine Kodierung auf Q basieren, während die zugrundeliegende Wahrscheinlichkeitsverteilung P ist.

Dementsprechen nutzt man $D_{KL}(P\|Q)$ oft im Kontext, wo P die tatsächliche Verteilung ist, Q unser Modell, dass wir geschätzt haben.

19.4 Kreuzentropie

Die Definition der Kreuzentropie ist nun

(292) $HK(X, P, Q) = H(X) + D(P\|Q)$

Äquivalent dazu gibt es (im diskreten Wahrscheinlichkeitsraum, den wir hier betrachten werden).

(293) $HK(X, P, Q) = -\sum_{x\in\Omega} P(X = x) \cdot log(Q(X = x))$

20 *No free lunch* – Gibt nix umsonst

20.1 Einleitung

Jeder Kurs über das maschinelle Lernen sollte die berühmten *no-free-lunch* (NFL) Theoreme behandeln, da sonst die Gefahr besteht, dass man grundsätzliche Dinge missversteht. NFL heißt soviel wie "gibt nix umsonst"; warum sie so heißen sollte bald klar werden. Diese Theorem sind von David Wolpert, dessen Papiere aber etwas sperrig sind.

Wir haben bis hierher viel über verschiedene Algorithmen besprochen, haben sehr elaborierte Methoden gesehen die in der Praxis sehr gut funktionieren. Das könnte uns zu dem Glauben verleiten, dass manche Algorithmen allgemein besser sind als andere. Die NFL-Theoreme besagen nun, dass genau das nicht stimmt. Auf einer allgemeinen Klasse von Lernproblemen lernen alle – wirklich alle, auch die sinnlosesten – Lernalgorithmen gleich gut. Und wenn ein Algorithmus auf einer Teilklasse von dieser Klasse besser funktioniert – z.B. der Teilklasse die wir beobachten – dann ist das nur deswegen, weil er auf einer anderen Teilklasse schlechter funktioniert.

Im Umkehrschluss bedeutet dass: wir können empirisch nie etwas über einen Algorithmus erfahren, sondern nur über diejenige Teilklasse unseres Problems, mit der wir getestet haben. Angewandtes maschinelles Lernen sagt uns also nur etwas über Phänomene, nicht über Algorithmen. Aber was genau heißen diese Begriffe, die wir eben benutzt haben? Das machen wir jetzt präzise.

20.2 Die NFL Theoreme und was sie bedeuten

Das Ziel (die objektive Funktion) Wir suchen eine Funktion

$$f : X \to Y$$

Das ist die objektive Funktion. Wolperts Ansatz ist allerdings etwas allgemeiner: er betrachtet **probabilistische Funktionen**, d.h. Funktionen, die für eine Eingabe $x \in X$ nicht ein $y \in Y$ ausgeben, sondern eine Wahrscheinlichkeitsverteilung

$$f(x) : Y \to [0, 1]$$

Natürlich ist leicht zu sehen, dass das nur eine Verallgemeinerung darstellt. Uns interessieren also Funktionen

$$f : (X \times Y) \to [0, 1]$$

so dass

$$\sum_{y \in Y} f(x, y) = 1, \text{ f.a. } x \in X$$

Wir bezeichnen die Menge dieser Verteilungen mit

$$prob(X^Y)$$

In unseren Daten finden wir allerdings keine Verteilungen, sondern tatsächlich Datenpunkte (x, y); also ist

$$D \subseteq X \times Y$$

Wir haben weiterhin die Annahme, dass alle Funktion $f \in prob(X^Y)$ apriori gleich wahrscheinlich sind – wir lernen also ohne Vorwissen. Algorithmen (bei uns) sind Funktionen, die Versuchen die richtige Funktion zu approximieren.

Kostenfunktion Wenn wir trainieren und bekommen ein falsches Ergebnis (unsere Vorhersage unterscheidet sich von der Beobachtung), dann kostet das, und dafür brauchen wir eine Kostenfunktion

$$k : Y \times Y \to \mathbb{R}.$$

Welche das ist, spielt keine Rolle, sie muss nur eine Bedingung erfüllen, nämlich sie muss **homogen** sein, das bedeutet: sei δ die Kronecker-delta Funktion definiert durch

$$\delta(x,y) = \begin{cases} 1, \text{ falls } x = y \\ 0 \text{ andernfalls} \end{cases} \tag{294}$$

Weiterhin bezeichnen wir

- mit c eine beliebige Konstante;
- mit y_f das "objektiv richtige $f(x)$ in unseren Daten D;
- mit y_h die Vorhersage unseres Algorithmus.

Nimm nun die Funktion

$$\delta(c, k(y_h, y_f)) : Y^3 \to \{0,1\} \tag{295}$$

in drei Argumenten, die uns 0 oder 1 liefert, je nachdem ob $k(x,y)$ den Wert c liefert. Wir können aus dieser Funktion nun das dritte Argument "herausmarginalisieren":

$$\Lambda(c, y_h) = \sum_{y_f \in Y} \delta(c, k(y_h, y_f)) : Y^2 \to \{0,1\} \tag{296}$$

Diese Funktion zählt, wie oft, für gegebenes c und y_h, die Distanz von beliebigen y_f und y_h den Wert c beträgt. Wir sagen k ist **homogen**, falls gilt: für beliebige y, y', c gilt

$$\Lambda(c, y) = \Lambda(c, y') \tag{297}$$

Das bedeutet: mit unserer Hypothese können wir keinen Einfluss darauf nehmen (unabhängig von der wahren, objektiven Funktion f), wie oft wir einen gewissen Kostenwert bekommen. Anders gesagt: in völliger Unkenntnis der objektiven Funktion wird unsere Kostenfunktion keine Hypothese bevorzugen. Das ist eine sehr allgemeine Bedingung die in der Praxis so gut wie immer erfüllt wird, da sie einfach besagt: es gibt nach Kostenfunktion keine bessere oder schlechtere Hypothesen, wenn wir noch keine Daten gesehen haben. Da die Funktion Λ, falls sie auf einer homogenen Kostenfunktion

basiert, dem zweiten Argument keine Rolle zumisst, können wir Λ als auch eine Funktion mit einem Argument auffassen, und schreiben in Zukunft

(298) $\Lambda(c)$

Eine einfache Funktion, die das nicht erfüllt, wäre z.B. eine Funktion

(299) $k(y_h, y_f) = max(\epsilon, y_h) \cdot (y_h - y_f)^2, \quad \epsilon > 0$

(das ϵ ist da um einen trivial beste Hypothese $y_h = 0$ zu verhindern). Diese Funktion bevorzugt kleine y_h unabhängig von y_f, ist also voreingenommen.

Stichproben sammeln Wir nehmen an, es gibt eine

$$\text{Wahrscheinlichkeitsverteilung } \theta : X \to [0, 1],$$

mittels derer wir Stichproben x aus X entnehmen, um dann schauen welchen Wert f dieser Stichprobe zuweist (wir beobachten also einen Punkt $(x, f(y))$). Das liegt unserem Training *und* unseren Tests zugrunde. Was wichtig ist, ist die Stichprobenwahrscheinlichkeit (likelihood) eines Datensatzes D, gegeben durch

$$P(D|f)$$

Diese Wahrscheinlichkeit hängt natürlich von θ ab. Wir sagen dass diese Verteilung (likelihood) **vertikal** ist, falls sie unabhängig ist von $f(x, y_f)$, falls x nicht in D vorkommt. Z.B. eine typische likelihood wäre gegeben durch

(300) $P(D|f) = \prod_{(x,y)\in D} \theta(x) f(x, y)$

(das wäre IID sampling). Wir brauchen Stichproben sowohl zum Testen als auch zum Training, und wir nehmen an das beide Male dieselbe Verteilung und Methode zugrunde liegt (das ist im Normalfall gewährleistet).

Der Algorithmus und sein Training Was wir weiterhing brauchen ist der Algorithmus a. Der Algorithmus ändert in jedem Schritt seine vorhersagen: initialisiert ist er eine Funktion

$$a_0 \in X^Y.$$

Nun ist es wichtig, dass der Algorithmus seine Daten in einer bestimmte Reihenfolge bekommt. Wenn er Kosten k_1 für x_1 und seine Vorhersage $a_0(x_1)$ bekommt:

(301) $k_1 := k(y_1, a_0(x_1))$

dann ändert er sich zu $a_1 \in X^Y$ usw. Der Algorithmus ist erstmal eine Folge von Funktionen

$$\langle a_0 : X \to Y, a_1 : X \to Y, a_2 : X \to Y, ... \rangle$$

Wir geben dem Algorithmus das Argument x_1, berechnen Kosten k_1, liefern diesen Wert als Eingabe (backpropagation), und dann bekommen wir a_1. Als nächstes bekommen wir

(302) $k_2 = d(a_1(x_2), f(x_2))$

Das ist **Training** unseres Algorithmus auf unseren Daten D.

Wahrscheinlichkeit von Kosten im Test Wor können nun definieren, was die Wahrscheinlichkeit ist, gewisse Kosten zu haben, gegeben einen Datensatz D. Wir fassen dafür k als eine Zufallsvariable auf:

$$P(k = x | f, a, D)$$

bezeichnet die Wahrscheinlichkeit, im nächsten gesammelten Beispiel von der zugrundeliegenden Verteilung f, mit Algorithmus a trainiert auf Datensatz D, die Kosten x zu haben. Z.B.

(303) $P(k = 0 | f, a, D)$

Das ist die Wahrscheinlichkeit, das nächste Beispiel richtig zu raten. Wie ist diese Funktion genau definiert? Das hängt von unserem sampling ab, im IID-Fall wäre das wie folgt:

(304) $$P(k = z | f, a, D) = \sum_{L(y, a_D(x)) = z} \theta(x) f(x, y)$$

Also die Wahrscheinlichkeit, das wir ein x ziehen, mal die Wahrscheinlichkeit dass $f(x) = y$, für alle (x, y) so dass $k(a_D(x), y) = z$, und das aufsummiert. (Die Bedingung: f, D muss konsistent sein?)

Einige NFL Theoreme Wichtig ist: alle Theoreme beziehen sich auf *off training set error* (OTS), also der Fehler wird nur auf Datenpunkten gemessen, die nicht im unserem Trainingsset vorkamen. Andernfalls würde das Ergebnis nicht gelten, den auf diesen Punkten können wir sehr leicht bessere/schlechtere Algorithmen ausmachen.

Das erste Wolpertsche NFL Theorem besagt (grob):

Theorem 4 (NFL 1) *Sei Λ wie oben für eine homogene Kostenfunktion definiert. Dann gilt: der normalisierte Durchschnitt über alle f im Kandidatenraum von $P(k = x|f, a, D)$ ist gleich $\Lambda(x)/r$ für ein Konstante r, unabhängig von a. Also insbesondere: für beliebige Algorithmen $a1, a2$ gilt:*

$$\sum_{f \in prob(X^Y)} P(k = z|f, D, a1) = \sum_{f \in prob(X^Y)} P(k = z|f, D, a2)$$

Das bedeutet: die Wahrscheinlichkeit, dass wir mit einem Algorithmus, gegebenen Trainingsset in Unkenntnis der objektiven Funktion als nächstes einen Fehler von k haben (z.B. $k = 0$, also richtig zu liegen), ist für *jeden Algorithus gleich*, und eine einfache Funktion über unsere Kostenfunktion; der Algorithmus ist irrelevant.

Der nächste Schritt besteht darin, das ganze unabhängig zu machen von D – der Datensatz soll also nicht gegeben sein; stattdessen nur seine Größe. Das Problem dabei ist, dass verschiedene fs unterschiedlich beitragen, da sie den Daten unterschiedliche Wahrscheinlichkeiten zuweisen. Wir haben:

$$P(k = z|a, f, |D| = m) = \sum_{|D|=m} P(k = z|f, a, D)P(D|f) \quad (305)$$

Das folgt aus den normalen Regeln zur Marginalisierung, und $P(D|f)$ ist *nicht* uniform über f (die likelihood der Daten ändert sich mit der objektiven Funktion). Dennoch gibt es auch hierfür ein NFL Theorem:

Theorem 5 (NFL 2) *Sei Λ wie oben für eine homogene Kostenfunktion definiert, und die Wahrscheinlichkeit $P(D|f)$ eine vertikale Verteilung für alle D, f. Dann gilt: der normalisierte Durchschnitt über alle f im Kandidatenraum von $P(k = x|f, a, |D| = m)$ ist gleich $\Lambda(x)/r$ für ein Konstante r, unabhängig von a. Also insbesondere: für beliebige Algorithmen $a1, a2$ gilt:*

$$\sum_{f \in prob(X^Y)} P(k = z|f, |D| = m, a1) = \sum_{f \in prob(X^Y)} P(k = z|f, |D| = m, a2)$$

Das bedeutet: auch wenn wir nur die Größe der Trainingsdaten als gegeben betrachten, sind alle Algorithmen gleich gut. Um das Ergebnis zu verstehen, muss man sehen dass f, gegeben im Term, ja die eigentliche Unbekannte ist. Wenn nun ein Algorithmus $a1$ auf einer Teilmenge (der Größe m) von $f \in X^Y$ konsisten nur geringere Kosten erzeugt als $a2$ (also konsistent besser ist), dann bedeutet dass das wir – vereinfacht gesprochen – eine entsprechende Funktion $f' \in prob(X^Y)$ haben, auf der $a1$ *konsistent schlechter* abschneidet.

Permutierbare Funktionen – ein notwendiges& hinreichendes Kriterium Man kann dieses Theorem noch etwas stärker formulieren: die Bedingung $f \in X^Y$ (also der gesamte Funktionenraum ist involviert) ist etwas zu schwach. Eine **Permutation** ist eine Funktion

$$\pi : M \to M, \text{ wobei } \pi^{-1} \circ \pi(x) = x$$

Also eine Bijektion auf einer Menge. Sei $F \subseteq X^Y$ eine Menge von Funktionen. Wir sagen diese Menge ist *geschlossen unter Permutation*, wenn folgende Bedingung erfüllt wird:

$$f \in F \Longrightarrow f \circ \pi \in F$$

Also wir können Eingaben beliebig permutieren, wir bleiben in der Klasse. Das erlaubt es uns z.B. den Ausgaberaum beliebig einzuschränken. Als Beispiel nimm folgende Klasse: sei $X = \mathbb{R} = Y$. Dann ist

- $F_{mon} \subseteq X^Y$, die klasse der monotonen Funktionen, ist nicht geschlossen unter Permutation (setze $\pi(n) = n + 1$ falls n gerade), $\pi(n) = n - 1$ falls n ungerade, $\pi(x) = x$ falls $n \notin \natural$.

- Die Klasse der linearen Funktionen (oder Polynome) ist nicht geschlossen unter Permutation (nimm obiges Beispiel, das macht die Funktion unstetig)
- Die Klasse der Wahrscheinlichkeitsfunktion $f : X \to [0, 1]$ ist geschlossen unter Permutation.

Der formale Abschluss unter Permutation hat eine intuitive Bedeutung: ein Funktionenraum ist geschlossen unter Permutation, wenn wir in gewissem Sinne keine *apriori*-Aussage über Eingaben – Ausgabe Relationen machen können. Nun lässt sich das Theorem stärker formulieren:

Theorem 6 (NFL3) *Für zwei beliebige Algorithmen a_1, a_2 gilt:*

$$\sum_{f \in F} P(k = z|f, |D| = m, a_1) = \sum_{f \in F} P(k = z|f, |D| = m, a_2)$$

genau dann wenn F unter Permutation abgeschlossen ist.

Der Abschluss unter Permutation ist also hinreichend und notwendig, damit die NFL Theoreme gelten. Ähnlich lassen sich andere Theoreme stärker formulieren, wir lassen das aber. Diese Theoreme sind sehr robust, und gelten noch für eine ganze Reihe weiterer Fälle. Das sog. zweite NFL Theorem ist aber z.B. wesentlich komplexer und nimmt an, dass sich die objektive Funktion über die Zeit ändert. Das macht natürlich einen Unterschied, da die Tatsache, dass wir eine gewissen Teil als Daten zuerst sehen, natürlich einen Unterschied, da unsere Funktion später sich anders verhalten kann. Auch diese Erweiterung ändert also nichts an den Ergebnissen.

20.3 Was bedeutet das also...

... wenn eine Klasse von Lernalgorithmen (z.B. neuronale Netze) auf einer gewissen Klasse von Problemen gut funktioniert? Man darf die NFL Theoreme nicht dahingehend missverstehen, dass das reiner Zufall wäre, das wäre praktisch ausgeschlossen. Stattdessen bedeutet das soviel wie: wir haben eine Teilklasse von Funktionen, die nicht unter Permutation geschlossen sind, anders gesagt, wir machen gewisse *apriori* Annahmen über unseren möglichen Funktionenraum. Zu sagen, dass ein Ansatz gut funktioniert, bedeutet: zu sagen, dass unser Funktionenraum auf eine gewisse Weise von vornherein eingeschränkt ist. Wir machen also Annahmen über unsere möglichen objektiven Funktionen. Das bedeutet, anders gesagt: effektives Lernen ist nur

möglich, wenn wir starke Annahmen machen über was wir lernen wollen. Es gibt aber umgekehrt keine Möglichkeit, objektiv zu prüfen ob diese Annahmen richtig sind.

20.4 NFL und Probabilistische Optimierung

Es gibt einen interessanten Zusammenhang zwischen NFL und probabilistischer Optimierung. Unter probabilistischer Optimierung verstehen wir hier folgendes: wir nehmen an, wir haben ein diskretes Klassifikationsproblem, sagen wir Einfachheit halber

$$f : M \to \{0, 1\}$$

Dieses Problem hat eine gewisse Zeitkomplexität. Bsp. dafür sind:

- Gegeben eine CFG G und ein Wort w, ist $w \in L(G)$?
- Ist eine gewisse Formel α der Aussagen/Prädikatenlogik eine Tautologie? (NP vollständig)

Nun gibt es verschiedene Methoden, diese Probleme anzugehen; eine Methode wäre, Wahrscheinlichkeiten zu nutzen (z.B. nutzen wir eine Strategie wie *random walk*). Das kann mitunter sehr effektiv sein. Die NFL Theoreme besagen nun: wann immer eine gewisse probabilistische Strategie auf einer Teilklasse von M besser arbeitet, wird sie auf einer anderen Teilklasse schlechter abschneiden (vorausgesetzt M ist abgeschlossen unter Permutation in einem gewissen Sinn). Das bedeutet: alle probabilistischen Methoden sind im Mittel gleich gut.

21 Kleines Wörterbuch der technischen Begriffe

Körper sind kompliziert zu definieren (9 Axiome); es reicht zu wissen dass es nur drei bekannte Körper gibt, die rationalen, die reellen und die komplexen Zahlen. Wir bleiben hier bei den reellen Zahlen, von daher gibt es keine weiteren Probleme. Wenn wir Vektoren oder Matrizen betrachten, dann sind diese immer über Körper konstruiert, d.h. die einzelnen Komponenten sind normalerweise reelle Zahlen. Man spricht in diesem Zusammenhang auch oft Skalaren, was nichts anderes ist als eine reelle Zahl, allgemeiner: ein Objekt aus dem Körper, über den wir Vektoren, Matrizen etc. konstruieren.

Vektoren und Vektorraum, konkret : Sei K ein Körper, $n \in \mathbb{N}$ eine beliebige natürliche Zahl. Ein Vektorraum V ist eine Teilmenge

$$V \subseteq K^n,$$

die geschlossen ist unter gewissen Operationen, die wir gleich definieren. Ein Vektor

$$\vec{v} \in V$$

ist also für uns einfach ein Tupel aus n reellen Zahlen. Wir definieren das Skalarprodukt

$$\lambda \cdot (v_1, ..., v_n), \text{ wobei } \lambda \in \mathbb{R},$$

mit

(306) $\lambda \cdot (v_1, ..., v_n) = (\lambda v_1, ..., \lambda v_n)$

Die Addition zweier Vektoren ist definiert durch:

(307) $(v_1, ..., v_n) + (w_1, ..., w_n) = (v_1 + w_1, ..., v_n + w_n)$

Ein Vektorraum ist nun wie folgt definiert:

1. V ist abgeschlossen unter $+$ und $\cdot$ mit allen Skalaren.
2. V enthält 0_V, den sog. Nullvektor so dass $\vec{v} + 0_V = \vec{v}$.
3. V enthält für jeden Vektor $\vec{v}$ ein inverses Element $-\vec{v}$, so dass $\vec{v}+(-\vec{v}) = 0_V$. Natürlich ist leicht zu sehen dass $-\vec{v} = (-1) \cdot \vec{v}$.

Der wichtigste Begriff der einfachen Vektorrechnung ist der der **linearen Abbildung**: eine lineare Abbildung ist eine Funktion $f : V \to V$ so dass $f(\vec{v}) = (\lambda \cdot \vec{v}) + \vec{w}$, wobei λ ein Skalar ist, $\vec{w} \in V$. Falls V ein Vektorraum ist, f eine lineare Abbildung, dann ist auch $f[V] = \{f(\vec{v}] : \vec{v} \in V\}$ ein Vektorraum.

Hyperebenen stellen eine Generalisierung von Ebenen (im 3-dimensionalen Raum) auf beliebige Dimensionen dar. Eine normale Ebene ist spezifiert durch einen **Stützvektor** $\vec{s}$ und zwei linear unabhängige **Richtungsvektoren** $\vec{r}_1, \}, \vec{r}_2$. Ein Punkt p liegt auf der Ebene, falls er sich darstellen lässt als $p = \lambda_1\vec{r}_1 + \lambda_2\vec{r}_2 + \vec{s}$. Diese Definition lässt sich leicht verallgemeinern: man nimmt, für einen Raum $\mathbb{R}^n$, einfach den Stützvektor $\vec{s} \in \mathbb{R}^n$ und ebenso $n-1$ linear unabhängige Richtungsvektoren $\vec{r}_1, ..., \vec{r}_{n-1}$. Technisch gesehen ist die Hyperebene eine Menge von Punkten:

$$H = \{\vec{s} + \lambda_1\vec{r}_1 + ... + \lambda_{n-1}\vec{r}_{n-1} : \lambda_1, ..., \lambda_{n-1} \in \mathbb{R}\} \tag{308}$$

Norm : definiert auf einem Vektorraum V ist das eine Funktion $\| - \| : V \to \mathbb{R}_0^+$; es werden also beliebige Vektoren auf einen nicht-negativen Wert abgebildet. Zusätzlich muss $\| - \|$ noch folgende Bedingungen erfüllen f.a. $\vec{v} \in V$, $\lambda \in \mathbb{R}$.

1. $\|\vec{v}\| = 0 \Rightarrow \vec{v} = 0_V$
2. $\|\lambda \cdot \vec{v}\| = |\lambda| \cdot \|\vec{v}\|$, wobei $|\lambda|$ der Betrag ist
3. $\|\vec{v} + \vec{w}\| \leq \|\vec{v}\| + \|\vec{w}\|$

Die intuitivste Norm ist die *euklidische*, die jedem Vektor seine **Länge** zuweise (wenn wir einen Vektor als eine Linie vom Ursprung auf seine Koordinaten (im n-dimensionalen Raum) auffassen. Diese Norm basiert auf einer Verallgemeinerung des Satz des Pythagoras:

$$\|(v_1, ..., v_n)\| = \sqrt{v_1^2 + ... + v_n^2}$$

In dieser geometrischen Interpretation wird Bedingung 3 zur **Dreiecksungleichung**: in jedem rechwinkligen Dreieck ist die Länge der Hypothenuse geringer als die Summe der Länge der Katheten. Es gibt aber noch viele weitere Normen, z.B. die sog. p-Norm, wobei $p \geq 1$ eine reelle Zahl ist:

$$\|(v_1, ..., v_n)\|_p = (\sum_{i=1}^{n} |v_i|^p)^{\frac{1}{p}}$$

Für $p = 1$ vereinfacht sich das zu

$$\|(v_1, ..., v_n)\|_1 = \sum_{i=1}^{n} |v_i| \quad (309)$$

Das ist die sog. Manhatten-Norm, weil man immer rechtwinklig um die Blocks fahren muss – das gibt im 2-dimensionalen Fall also die kürzeste Strecke in Manhattan an.

Metrik Darauf basiert der Begriff der Metrik; jede Norm induziert eine Metrik d mittels

$$d(\vec{v}, \vec{w}) = \|\vec{v} - \vec{w}\| \quad (310)$$

wobei natürlich

$$\vec{v} - \vec{w} = (v_1 - w_1, ..., v_i - w_i) \quad (311)$$

Es ist nicht schwer zu sehen dass gilt:

- $d(x, y) \geq 0$ (positiv)
- $d(x, y) = d(y, x)$ (symmetrisch)
- $d(x, y) \leq d(x, z) + d(z, y)$ (Dreiecksungleichung)

Die **euklidische Distanz** ist definiert durch die euklidische Norm, mit

$$d_2(\vec{v}, \vec{w}) = \|\vec{v} - \vec{w}\|_2 \quad (312)$$

Kosinus (zweier Vektoren) Im rechtwinkligen Dreieck ist der Kosinus eines Winkels $cos(\alpha)$ definiert als

(Länge der Ankathete)/(Länge der Hypothenuse)

also $cos(90°) = 0$, $cos(0°) = 1$, und alles andere liegt dazwischen. Man kann den Kosinus auch verallgemeinert definieren als Funktion $cos : \mathbb{R} \to \mathbb{R}$, und zwar einigermaßen kompliziert als unendliche Reihe:

$$(313)\quad cos(x) = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n}}{2n!}$$

Was uns insbesondere interessiert ist der Kosinus zweier Vektoren. Im 2-dimensionalen Raum lässt sich das natürlich schön veranschaulichen in dem wir einfach einen Vektor als Hypothenuse auffassen, von seinem Ende eine Linie ziehen so dass sie den (verlängerten) anderen Vektor im rechten Winkel schneidet, und dann die geometrische Definition anwenden.

Skalarprodukt : Bislang haben wir Multiplikation nur mit Skalaren ausgeführt; es gab noch keine Möglichkeit, zwei Vektoren miteinander zu multiplizieren. Das macht das Skalarprodukt $\bullet : V^2 \to \mathbb{R}$. Wir multiplizieren also Vektoren und bekommen eine reelle Zahl (einen Skalar) als Ergebnis. Es gibt nicht das eine Skalarprodukt, sondern mehrere Arten es zu definieren. Wir brauchen hier nur die geläufigste, nämlich das Standard-Skalarprodukt:

$$(314)\quad (a_1, ..., a_n)\top(b_1, ..., b_n) = a_1b_1 + a_2b_2 + ... + a_nb_n$$

Wenn wir Vektoren als spezielle Matrizen betrachten, ist das Standard-Skalarprodukt nur das Produkt zweier Matrizen, nämlich der Transposition der ersten und der zweiten. Wichtig ist dass wir die erste Matrix transponieren, sonst wäre das Produkt undefiniert; weiterhin setzt diese Multiplikation voraus, dass beide Vektoren die gleiche Länge haben.

Hadamard Produkt Das ist eine Funktion $\odot : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, einfach definiert durch **punktweise Multplikation**: $(x_1, ..., x_n) \odot (y_1, ..., y_n) = (x_1y_1, ..., x_ny_n)$. Es ist also parallel zur Vektoraddition, nur mit Multiplikation.

Matrix Matrizen generalisieren Vektoren in dem Sinne dass ein Vektor $\vec{v} \in \mathbb{R}^n$ ist, eine Matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ eine kompliziertere Struktur ist, bzw. ein Vektor ist eine Matrix mit $m = 1$. Matrizen schreibt man wie folgt: $\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{pmatrix}$ Zeilen zuerst, dann Spalten! Um ein Objekt an einer bestimmten Koordinate Matrix zu denotieren schreiben wir $\mathbf{A}_{ij}$; z.B. im obigen Beispiel $\mathbf{A}_{23} = b_3$.

Rechnen mit Matrizen ist etwas gewöhnungbedürftig; wir haben normalerweise die Addition und die Multiplikation. Addition ist im Prinzip dasselbe wie bei Vektoren. Nimm zwei Matrizen $\mathbf{A}, \mathbf{B} \subseteq \mathbb{R}^{m \times n}$, dann haben wir $\mathbf{A} + \mathbf{B} \subseteq \mathbb{R}^{m \times n}$, und $(\mathbf{A} + \mathbf{B})_{ij} = \mathbf{A}_{ij} + \mathbf{B}_{ij}$. Also ist die Addition punktweise definiert, genau dann wenn die beiden Matrizen dieselben Maße haben.

Die Multiplikation ist etwas komplizierter. Nimm zwei Matrizen $\mathbf{A} \in \mathbb{R}^{m \times p}$, $\mathbf{B} \in \mathbb{R}^{p \times n}$, wobei wichtig ist das die beiden ps identisch sind. Wir definieren dann

$(\mathbf{A} \cdot \mathbf{B})_{ij} = \sum_{k=1}^{p} (A_{ik} \cdot B_{kj})$.

Diese Multiplikation ist natürlich *nicht* kommutativ; um das zu sehen, nehmen wir an $p = 1$, Matrizen $\mathbf{A} \in \mathbb{R}3 \times 1$, $\mathbf{B} \in \mathbb{R}^{1 \times 3}$; dann haben wir:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix} = \begin{pmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & a_1 \cdot b_3 \\ a_2 \cdot b_1 & a_2 \cdot b_2 & a_3 \cdot b_3 \\ a_3 \cdot b_1 & a_3 \cdot b_2 & a_3 \cdot b_3 \end{pmatrix}$$

Wenn wir das umdrehen, dann bekommen wir:

$$\begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \big((a_1 \cdot b_1) + (a_2 \cdot b_2) + (a_3 \cdot b_3)\big)$$

Also bekommen wir in diesem Fall eine 1×1-dimensionale Matrix. Es kann natürlich auch oft vorkommen dass $\mathbf{A} \cdot \mathbf{B}$ definiert ist, während $\mathbf{A} \cdot \mathbf{B}$ undefiniert bleibt. Ein wichtiges Konzept in dieser Hinsicht ist die **Einheitsmatrix**, definiert als:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & \ddots & 0 \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

Diese Matrix ist also quadratisch, mit allen Werten 0, nur in der Diagonale

hat sie den Wert 1. Einheitsmatrizen gibt es für jedes $n \in \mathbb{N}$, denn die Größe muss natürlich passen; wir denotieren die $n \times n$ Einheitsmatrix mit $\mathbf{1}_n$. Dann ist leicht zu sehen: Sei $\mathbf{A} \in \mathbb{R}m \times n$ eine beliebige Matrix. Dann ist

$\mathbf{A} \cdot 1_n = 1_m \cdot \mathbf{A} = \mathbf{A}$

Eigenvektor und Eigenwert

Gradienten Gradienten sind eine Generalisierung von Ableitungen für multivariate Funktionen. Ableitungen für Funktionen $f : \mathbb{R} \to \mathbb{R}$ sollten bekannt sein; intuitiv gibt der Wert der Ableitung $\frac{d}{d(x)}f$ an jedem Punkt den Grad an, in dem die Funktion zu- bzw. abnimmt. Wenn man nun eine multivariate Funktion hat, dann kann man sie nach jeder Variable *partiell ableiten*. Sei $f : \mathbb{R}^n \to \mathbb{R}$ eine Funktion, $i \in \{1, ..., n\}$; dann haben wir mit

$$\frac{df}{dx_i} f(x_1, ..., x_n) \tag{315}$$

die Ableitung nach x_i, die uns angibt, wie die Steigung der Funktion in der i-ten Dimension verläuft; die anderen Variablen sind natürlich Parameter dieser Steigung: sie geben an wir groß sie ist an diesem Punkt im Raum. Geometrisch betrachtet: sei $f(x, y)$ eine Funktion, die für zwei Koordinaten einer Oberfläche die jeweilige Höhe liefert. Die Ableitung nach x gibt nun an, wie groß die Steigung ist, wenn ich in Richtung x gehe. Dasselbe Prinzip für höhere Dimensionen.

Der Gradient von $f : \mathbb{R}^n \to \mathbb{R}$ ist eine Funktion $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$, dessen Ausgabekomponenten 1-n durch die partiellen Ableitungen von f nach $x_1, ..., x_n$ berechnet werden, d.h.

$$\nabla f(x_1, ..., x_n) = (\frac{df}{dx_1} f(x_1, ..., x_n), ..., \frac{df}{dx_n} f(x_1, ..., x_n)) \tag{316}$$

Alternativ kann man auch die Einheitsvektoren zur Definition nutzen; nenne die n-dimensionalen Einheitsvektoren mit $i$ter Komponente 1 $1_1, ..., 1_n$; dann ist

$$\nabla f(x_1, ..., x_n) = \frac{df}{dx_1} f(x_1, ..., x_n)1_1 + ... + \frac{df}{dx_n} f(x_1, ..., x_n)1_n \tag{317}$$

Die geometrische Interpretation des Gradienten ist folgende: der Gradient einer Funktion gibt einen Vektor, und der zeigt in diejenige Richtung, in der

die Funktion am schnellsten steigt. Also im Fall $f : \mathbb{R}^n \to \mathbb{R}$: die Richtung in der Ebene, in die man gehen muss um am schnellsten Höhe zu gewinnen.

Es ist klar, dass man, um ein *lokales* Maximum einer Funktion zu finden, einfach nur dem Gradienten "nachgehen" muss (mit einer gewissen *Lernrate* ϵ; wir haben das Maximum gefunden, falls der Gradient 0 an allen Stellen ist.

Lernrate Eine kleine Zahl $l > 0$, die besagt, wie wir den Wert der Funktion verändern. Was normalerweise wissen ist: wir kennen den Wert $f(x)$, und wir wissen dass für ein $\epsilon > 0$, $f(x+\epsilon) < f(x)$, oder $f(x-\epsilon) < f(x)$. Was wir nicht kennen ist ϵ. Deswegen legen wir die Lernrate l fest, die besagt, wie gross die Schritte sein sollen, die wir in die richtige Richtung gehen. Wenn l zu klein ist, brauchen wir länger (mehr Schritte) als nötig; wenn l zu groß ist, dann kann es sein dass wir den optimalen Punkt verpassen.