

Neuronale Netze und Tiefe Architekturen

Christian Wurm
cwurm@phil.hhu.de

February 5, 2021

Abstract

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

Contents

1	Maschinelles Lernen: Klassifikation und Regression	6
1.1	Test und Trainingsdaten, Overfitting und Underfitting	12
2	Maschinelles Lernen – Überblick	17
2.1	Was brauchen wir?	17
3	Vorspiel 0: Grundlagen der Wahrscheinlichkeitstheorie	19
3.1	Bedeutung von Wahrscheinlichkeiten	19
3.2	Desiderata	19
3.3	Boolesche Algebren	19
3.4	Einige Beobachtungen	20
3.5	Definition von Wahrscheinlichkeitsräumen	22
3.6	Ereignisse und Ergebnisse	23
3.7	Die Komplement-Regel	23
3.8	Die Summenregel	24
3.9	Die Produktregel	24
3.10	Das sog. Bayessche Gesetz und seine Bedeutung	25

4	Exkurs: Conditional Random Fields	26
4.1	Definition	26
4.2	Ein Beispiel	28
4.3	Parameter schätzen	28
5	Lineare Modelle	29
5.1	Die einfache lineare Regression	29
5.2	Der komplexe lineare Fall	31
5.3	Lineare Regression in \mathbb{R}	32
5.4	Beschränkungen linearer Modelle, lineare Separierbarkeit . . .	34
5.5	Abschluss unter Komposition	36
5.6	Höhere Dimensionalität	38
6	Die Logistische Regression und das einfache Perzeptron	41
6.1	Das einfache Perzeptron	41
6.2	Bedeutung	43
6.3	Exkurs: Lernen & Anwendung von einfachen Perzeptronen . .	46
6.4	Höherdimensionale Modelle mit nichtlinearen Funktionen . . .	49
6.5	Exkurs: nearest neighbour Regression	50
7	Nichtlineare Funktionen: Aktivierung u.a.	53
7.1	Überblick	53
7.2	Innere Funktionen (Aktivierungsfunktionen)	55
7.3	Punktweise Erweiterung von Funktionen	56
7.4	Rectified linear units und ihre Generalisierung	57
7.5	Ausgabefunktionen	59
8	Was können Tiefe Architekturen?	61
8.1	Mächtigkeit ist nicht Lernbarkeit	61
8.2	Universelle Approximation	63
8.3	Boolesche Funktionen: Ein Beispiel	66
9	Was <i>lernen</i> Tiefe Architekturen?	68
9.1	Parameter und Freiheitsgrade	68
9.2	(Boolesche) Entscheidungsfunktionen	70
9.3	Entscheidungsbäume	72
9.4	Neuronale Netze als Entscheidungsbäume	74

10 Optimierung von Neuronalen Netzen	76
10.1 Vorspiel 1: Analytische Optimierung	76
10.2 Vorspiel 2: Konvexe Optimierung	77
11 Gradienten berechnen	79
12 Gradientenbasierte Optimierung	80
12.1 Das Problem in höheren Dimensionen	80
12.2 Die effektive Prozedur	82
13 Differenzieren – einige Faustregeln	84
14 Tiefe Netze trainieren	87
14.1 Parameter und Hyperparameter	87
14.2 Backpropagation	91
14.3 Die Fehlerfunktion	92
14.4 Backpropagation Übersicht	93
14.5 Berechnungsgraphen	94
14.6 Backpropagation – Verwaltung	95
14.7 Berechnungsgraphen – Knoten und Funktionen	96
14.8 Backpropagation – Der Feedforward-Step	97
14.9 Backward-propagation	99
14.10 Backpropagation: Ausgabe	101
14.11 Beweis von Satz 11	102
14.12 Gradient descent	105
14.13 Alternative: Momentum	106
14.14 Zusammenfassung	107
15 Initialisierung	108
16 Regularisierung	109
17 Word embeddings	110
17.1 Einleitung	110
17.2 Skip-gram models	111
17.3 Hierarchical Softmax	116
17.4 Sog. “Additive Kompositionalität”	118

18 Recurrent Neural Networks	119
18.1 Motivation	119
18.2 RNN – zwei Auffassungen	119
18.3 Definition	120
18.4 Das <i>bag of words</i> -Modell CBOW	123
18.5 SRNN	124
18.6 RNNs stapeln	125
18.7 Training von RNNs	127
18.8 Vanishing Gradient	129
19 Zwei Anwendungsbeispiele für RNN	130
19.1 RNN für <i>sequence labelling</i>	131
19.2 RNNs als Sprachmodelle	135
20 Architekturen mit <i>gates</i> – LSTM	138
20.1 Vorspiel	138
20.2 LSTM – Definition	141
20.3 Gated recurrent units	144
21 Convolutional Neural Networks	146
21.1 Convolution – was ist das?	146
21.2 Convolution – Beispiel 1	147
21.3 Convolution – Beispiel 2	151
21.4 Motivation	156
22 Kostenfunktionen – intern	157
22.1 Entropie etc.	159
22.2 Bedingte Entropie	162
22.3 Kullback-Leibler-Divergenz	163
22.4 Kreuzentropie	164
22.5 Kostenfunktion für binäre Klassifizierung	165
22.6 Kostenfunktion für beliebige Klassifizierung	166
23 Sequence-to-sequence Modelle	167
23.1 Einfache Modelle	167
23.2 Bidirektionale Modelle	168
23.3 Vector to sequence (VtS)	169
23.4 Encoder – decoder (ED)	171

24 Attention	172
24.1 Intuition: Alinierung und Wahrscheinlichkeit	172
24.2 Definition - Die erste Architektur	174
24.3 Training	176
24.4 Attention is all you need – der Transformer	177
25 Transfer Learning – Fine-tuning	179
26 Metriken und Kostenfunktionen – extern	180
27 <i>No free lunch</i> – Gibt nix umsonst	182
27.1 Einleitung	182
27.2 Die NFL Theoreme und was sie bedeuten	183
27.3 Einige NFL Theoreme	188
27.4 Was bedeutet das also...	191
27.5 NFL und Probabilistische Optimierung	192
28 Übersicht: eine Art “pipeline”	193
29 Kleines Wörterbuch der linearen Algebra (und etwas mehr)	195

1 Maschinelles Lernen: Klassifikation und Regression

Was man im maschinellen Lernen “lernen” oder besser gesagt *approximieren* will sind normalerweise Funktionen. Das bedeutet: wir versuchen, dem Verhalten einer “wahren” Funktion, von der wir nur eine gewisse endliche Anzahl an Datenpunkten kennen, möglichst nahe zu kommen mit unserem Modell. Wichtig ist: die wahre Funktion, die wir approximieren möchten, ist und bleibt unbekannt. Ob das, was wir gelernt haben, wirklich adäquat ist, können wir nur näherungsweise sagen.

Klassifikation, Regression Der Begriff der Funktionenapproximation ist sehr allgemein, und es lässt sich normalerweise alles darunter packen. Man unterscheidet aber zwei wichtige Fälle nämlich Klassifikation und Regression. (Man unterscheidet die beiden *einzig anhand des Wertebereichs der Zielfunktion*:

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

Im Deep Learning geht es normalerweise um Klassifikationsaufgaben: wir möchten für eine unendliche Menge von möglichen Eingaben eine endliche Menge von Ausgaben haben. Wichtig hierbei ist: Neuronale Netze klassifizieren mit Mitteln der Regression. Jedes neuronale Netz ist erstmal eine Funktion $N : \mathbb{R}^n \rightarrow \mathbb{R}^m$, ohne Ausnahme. Um daraus einen Klassifikator zu bauen, müssen wir die numerische Ausgabe uminterpretieren, und evtl. auch die nicht-numerische Eingabe in einen Vektor kodieren. Neuronale Netze haben übrigens noch eine dritte Möglichkeit, die sonst eher marginal vorkommt, in der Computerlinguistik dagegen sehr häufig:

3. Wir approximieren eine Funktion $f : \Sigma^* \rightarrow T^*$ (sequence-to-sequence labelling). Das steht zwischen den beiden obigen Modellen, denn die Menge der möglichen Ausgaben ist unendlich, aber nicht reellwertig.

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, maschinelles Lernen 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 Funktion, die gut approximiert, ist die beste. Das ist aber nur in manchen Fällen ein sinnvolles Kriterium, weil es viele verschiedene Definitionen von “einfach” gibt. Man kann eigentlich nur sagen: ob eine Approximation (sprich: ein Modell appliziert auf ein Problem) 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.

Neuronale Netze haben aber in den letzten Jahren auf praktisch allen Feldern der künstlichen Intelligenz durchschlagende Erfolge erzielt. Warum das so ist weiß eigentlich niemand genau; wir werden versuchen diese Tatsache hier ein wenig zu beleuchten, aber letztlich ist das ein empirisches Phänomen, das man nur im Nachhinein versuchen kann zu erklären.

Nimm z.B. einen Datensatz

$$D \subseteq \mathbb{R} \times \mathbb{R} \times \mathbb{R}$$

Wir haben also z.B. $D = \{(2, 6.4, 3.92), (4.9, 3.57, 2.51), \dots\}$. Es stellt sich erstmal die Frage: **was wollen wir daraus lernen?** Hier gibt es verschiedene Möglichkeiten. Eine Antwort wäre:

Die Aufgabe Wir möchten die dritte Komponente vorhersagen, gegeben die ersten beiden. Das bedeutet: wir möchten approximieren eine Funktion

$$f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

und zwar anhand einer Menge von Datenpunkten: $(2, 6.4, 3.92) \in D$ bedeutet dann soviel wie:

$$(1) \quad f(2, 6.4) = 3.92 \text{ (bzw. } f(2, 6.4) \approx 3.92)$$

Sprich: jeder Datenpunkt ist eine **Instanz** der Funktion. Wir versuchen die komplette Funktion die rekonstruieren, oder bzw. besser: eben zu approximieren (die Gründe hierfür sind verschieden: wir möchten nicht unbedingt, dass unsere Rekonstruktion genau mit den Instanzen übereinstimmt, denn 1. kann das zu overfitting führen, und 2. haben wir oft Rauschen/Störungen in den Daten).

Also nochmal kompakt:

1. Wir nehmen einen Datensatz (eine endliche Relation, normalerweise)
2. Wir entscheiden welche Komponenten die Argumente der Funktion sind, welches die Werte
3. Auf diese Art bekommen wir die *Instanzen* einer Funktion, der Form $f(a_i) = b_i$
4. Diese zugrundeliegende Funktion versuchen wir dann, mit unseren Mitteln, zu approximieren.

Das nennt man maschinelles Lernen.

Hypothesenraum Das Grundproblem ist dass wir f normalerweise nicht kennen, d.h. wir können nie wissen, ob unsere Induktion erfolgreich war oder nicht. Wir werden die Zielfunktion f auch niemals kennenlernen. Deswegen spielt f keine wichtige Rolle, und alles, was wir vorschlagen, sind immer nur **Hypothesen**. Wir nennen unsere Hypothesen h , und natürlich, falls wir eine Funktion $f : M \rightarrow N$ approximieren wollen, dann haben wir

$$h : M \rightarrow N$$

Alles was wir wissen können ist ob bzw. wie gut h übereinstimmt mit f auf dem endlichen Datensatz D von Instanzen, den wir zur Verfügung haben. Umso wichtiger ist dafür er der sogenannte

Hypothesenraum H ,

d.i. eine Menge von möglichen Funktionen, aus der wir h auswählen. Der Raum \mathbf{H} ist durchaus nicht vorgegeben im Rahmen des Induktionsproblems, und die Wahl ist oft alles andere als einfach. Der Hypothesenraum entspricht, etwas moderner, dem **Modell**, das wir wählen, um unser Problem anzugehen. Jedes Modell hat seine Beschränkungen und Eigenarten, und legt daher die möglichen Hypothesen *a priori* fest!

Eine Beispielregression Das kann man sehr schön am Beispiel einer *Regression* sehen. Nehmen wir an, wir möchten eine Funktion

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

induzieren, z.B. um die Korrelation

$$\text{Tagestemperatur} \sim \text{Straftaten}$$

an einem gewissen Ort zu bestimmen (letztere gemittelt über einen längeren Zeitraum). Hier sieht man übrigens bereits, warum man nicht unbedingt will, dass f.a. Instanzen $f(a) = b$ auch gilt $h(a) = b$ – denn es handelt sich ja nur um Mittelwerte, und die Kriminalität wird ja von vielen anderen Faktoren beeinflusst, nur indirekt von der Temperatur.

Unser Datensatz D hat die Form

$$\begin{aligned} (10, 25.3), \text{ also } f(10) &= 25.3 \\ (14, 27.8), \text{ also } f(14) &= 27.8 \\ &\text{etc.} \end{aligned}$$

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 die **Konsistenzbedingung**: die Hypothese soll möglichst konsistent mit den Daten sein – je konsistenter, desto besser, denn das ist unser einziges gute Auswahlkriterium.

Und jetzt die Frage: was ist H , unser Hypothesenraum? Hier gibt es folgende Überlegungen:

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

Z.B.: sei

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

(ist einigermaßen konsistent mit den Datenpunkten). 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.

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

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 . Selbst wenn sie wirklich ernsthaft adäquater wäre, wäre Sie deutlich schlechter, einfach weil sie viel uninformativer ist. Das besagt also:

Wenn die richtige Hypothese darin ist, dann ist ein kleiner Hypothesenraum immer besser – sie ist dann einfach besser zu finden!

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

$$(4) \quad x^2 + ax + b$$

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

$$(5) \quad f(x) = ax + b + c \sin(x)$$

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!

1.1 Test und Trainingsdaten, Overfitting und Underfitting

Es gibt für das ML zwei große Probleme, oder besser gesagt: zwei Arten, wie unsere Ergebnisse danebenliegen können, nämlich

- *overfitting* und
- *underfitting*.

Underfitting Dies ist normalerweise weniger problematisch. Underfitting bedeutet einfach: unser Hypothesenraum ist zu klein. Es gibt keine Funktion in H , die das zugrundeliegende Muster von f ausreichend fassen kann. Underfitting kann man daran beobachten, dass es auf den Trainingsdaten einen ziemlich hohen Fehler gibt (dazu später mehr), als es gibt viele a_i so dass $f(a_i) \not\approx h(a_i)$. Deswegen ist dieser Fehler gut sichtbar.

Overfitting Das ist ein großes Problem im maschinellen Lernen, das uns permanent verfolgen wird, und das man nur mit ausgefeilten Methoden in den Griff bekommt. Overfitting kann man mit Üntergeneralisierung (!) übersetzen: wir generalisieren Dinge, die reiner Zufall sind. Anders gesagt: die Hypothese ist soz. maßgeschneidert für die Daten, und wird nicht ausreichend verallgemeinern und von irrelevanten Dingen abstrahieren.

Overfitting ist deswegen problematisch, weil man es nicht am Trainingsfehler sieht, ganz im Gegenteil: wenn das Modell overfittet, dann ist der Trainingsfehler sehr klein! Was man also macht ist folgendes:

- Man teilt den Datensatz in Test- und Trainingsdaten auf (20 zu 80)
- Man trainiert das Modell auf den Trainingsdaten, d.h. man setzt die Parameter des Modells
- Erst wenn das Training beendet ist, das Modell fertig “gelernt” hat, schaut man: wie groß ist der (durchschnittliche) Fehler auf den Testdaten?

Der entscheidende Punkt:

Wenn der Fehler auf den Testdaten deutlich größer ist also auf den Trainingsdaten, dann haben wir wahrscheinlich overfittet!

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 partitionieren unsere Daten in Test- und Trainingsatz, 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.

Deswegen ist es im Maschinellen Lernen unerlässlich, Test- und Trainingsdaten zu trennen. Allgemein gilt folgendes:

- Je mächtiger eine Modellklasse (größer der Hypothesenraum), desto eher laufen wir ins Overfitting.
- Je mächtiger eine Modellklasse (größer der Hypothesenraum), desto geringer wird der Fehler auf den Trainingsdaten sein. Aber: auf den Testdaten nicht unbedingt! Die Trennung erlaubt uns also, einfachere Modelle zu rehabilitieren - vorausgesetzt, sie underfitten nicht.
- Underfitting bedeutet: der Fehler sowohl auf Test- als auch Trainingsdaten ist groß.
- Im allgemeinen gilt: die Größe des Verhältnisses Parameter/Datenpunkte ist ein guter Indikator für das Risiko von Overfitting. Je kleiner der Wert ist, desto besser!

Ein Beispiel 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, das wir anwenden (mehr dazu gleich).

Besser kann man das aber anhand von Funktionsgraphen erklären (grob quadratische Korrelation, lineare Funktion underfittet, hohes Polynom overfittet).

Störparameter (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.)

Noch zwei kleine Punkte zur Aufteilung der Daten. Dabei gibt es ein paar Dinge zu beachten.

1. Wir müssen verhindern, dass diese Aufteilung sich ändert, während unser Algorithmus nicht auf 0 gesetzt wird – sonst passiert genau das, was man verhindern will: der Algorithmus sieht die Testdaten und kann auf ihnen Musterlösungen erzielen. Dann sind natürlich die gesamten Ergebnisse wertlos. Man muss also darauf achten dass die Aufteilung konstant ist. Da die Aufteilung zufällig erfolgt muss man die Zufallsparameter festsetzen.
2. Falls der Datensatz klein ist, muss man sichergehen, dass der Testsatz (20%) *repräsentativ* ist. Das heißt: alle Merkmale, die relevant sind, müssen auch vertreten sein. In diesem Fall müssen wir sicher sein, dass wir alle Merkmale vertreten. Das ist insbesondere schwierig bei kontinuierlichen Merkmalen: hier müssen wir kategorisieren, (sonst können wir keine sinnvollen Aufteilungen treffen; siehe Geron S.53)

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 Trainingsatz 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. Was man deswegen macht ist folgendes: wir machen von Anfang an eine **Dreiteilung der Daten**:

1. Trainingsdaten. Für ein gegebenes Modell werden hierauf die Parameter trainiert.
2. Development Set (Devset). Hierauf wird dann evaluiert. Wir machen das für verschiedene Hyperparameter, also verschiedene Modelle, und wählen damit das beste Modell aus und schließen Overfitting der Parameter aus.
3. Testdaten. Hierauf wird nun das beste Modell evaluiert. Das Ergebnis wird nun nicht mehr weiter benutzt um Modelle zu optimieren, sondern das Ergebnis auf den Testdaten ist *aussagekräftig*, um z.B. zu sagen, wie gut ein Modell ist.

Wir sehen also: sobald ein Datensatz genutzt wird, um irgendetwas (Parameter, Hyperparameter) zu optimieren, sind die Ergebnisse hierauf nicht mehr aussagekräftig was die Leistungsfähigkeit des Modells betrifft. Damit stellt sich die

Frage: Wie können wir Betrug verhindern? Das z.B. uns jemand Zahlen präsentiert (für die Qualität seines Modells), die auf solchem Overfitting basieren?

Antwort: alle Ergebnisse müssen immer replizierbar sein. Die Arbeit muss so dokumentiert sein, dass jeder das Modell (mit den Daten) und allen Trainingsschritten rekonstruieren kann. Dadurch ist natürlich Betrug ausgeschlossen, denn es reicht bereits eine Änderung der Zufallsparameter (für die Datenaufteilung), um ein etwaiges Overfitting zu entdecken.

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 “danebenliegt” 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

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

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 Beobachtungen, die wir haben, und unsere Kostenfunktion $K : \mathbf{Y} \times \mathbf{Y} \rightarrow \mathbb{R}$. 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

$$(6) \quad \underset{\theta \in \Theta}{\operatorname{argmin}} \sum_{(x,y) \in D} K(D(x), M(\theta)(x))$$

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

$$(7) \quad M(\theta)(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Also unser Modell ist ein Polynom. In diesem Fall sind $a_n, \dots, a_0 = \theta$ die Parameter der Funktion. Für K nehmen wir die Euklidische Distanz, also

$$(8) \quad K(x, y) = (x - y)^2$$

Dann bekommen wir

$$(9) \quad \sum_{(x,y) \in D} (a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 - y)^2$$

Das kann man ausmultiplizieren und bekommt dergestalt wiederum ein Polynom. Da 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.

(6) 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 (6) 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.

3 Vorspiel 0: Grundlagen der Wahrscheinlichkeitstheorie

3.1 Bedeutung von Wahrscheinlichkeiten

Eine Sache, die man gleich zu Anfang klären sollte, ist: *was bedeuten eigentlich Wahrscheinlichkeiten?* Üblicherweise ist man versucht zu sagen: wenn eine Münze mit einer Wahrscheinlichkeit von $1/2$ auf Kopf fällt, dann heißt das, dass sie perfekt symmetrisch ist, und weiterhin: wenn wir sie oft genug werfen, wird sie in ca. der Hälfte der Fälle auf Kopf landen. Die Wahrscheinlichkeit beschreibt also eine physische Eigenschaft und in der Folge ein Verhalten.

Das klingt gut, ist aber problematisch: was ist die Wahrscheinlichkeit, dass es Leben auf dem Mars gibt? Und gegeben dass wir eine Münze finden und werfen, was ist die Wahrscheinlichkeit, dass sie auf Kopf landet? Hier haben wir einen anderen Begriff von Wahrscheinlichkeit: er drückt die Stärke unserer Überzeugung aus. Diese kann – im Falle des Mars – mehr oder weniger informiert sein. Im Falle der Münze sagen wir: die Wahrscheinlichkeit, dass sie auf Kopf landet, ist $1/2$, denn wir haben keinerlei Wissen, dass uns dahin bringen würde, Kopf oder Zahl vorzuziehen. Diese uniforme Verteilung ist also **Ausdruck unserer Ignoranz**. Darauf beruht das Prinzip der Indifferenz.

3.2 Desiderata

Wir haben gesehen dass wir für die Wahrscheinlichkeitstheorie 2 große Desiderata haben:

1. Wir wollen (aussagen)logische Operationen für Ereignisse; und
2. wir möchten die logischen Operationen *numerisch interpretieren*, d.h. in numerische Funktionen verwandeln.

3.3 Boolesche Algebren

Logische Operationen können wir in Booleschen Algebren interpretieren:

Definition 1 Sei M eine Menge. Ein Mengensystem $\mathcal{M} \subseteq \wp(M)$ ist eine Boolesche Algebra über M , falls

1. $M \in \mathcal{M}, \emptyset \in \mathcal{M}$;
2. falls $N \in \mathcal{M}$, dann ist auch $\overline{N} := M - N \in \mathcal{M}$;
3. falls $N_1, N_2 \in \mathcal{M}$, dann sind auch $N_1 \cup N_2 \in \mathcal{M}$.

NB: die Definition impliziert dass falls $N_1, N_2 \in \mathcal{M}$, dann ist auch $N_1 \cap N_2 \in \mathcal{M}$, da $N_1 \cap N_2 = \overline{\overline{N_1} \cup \overline{N_2}}$. Unsere Definition betrifft eigentlich nur einen Spezialfall von Booleschen Algebren, nämlich solchen über Mengensystemen. Allerdings kann jede endliche Boolesche Algebra auf diesen Spezialfall reduziert werden.

Übung: Mengenlehre und Partitionen

- $M \cap N = \overline{\overline{M} \cup \overline{N}}$ (Interdefinierbarkeit 1)
- $M \cup N = \overline{\overline{M} \cap \overline{N}}$ (Interdefinierbarkeit 2)
- $\overline{\overline{M}} = M$ (doppeltes Komplement)
- $(M \cup N) \cap O = (M \cap O) \cup (N \cap O)$ (de Morgan)

Eine Partition einer Menge M ist eine Menge $X \subseteq \wp(M)$, d.h. $X = \{N_1, \dots, N_i\}$ (im endlichen Fall), und es gilt:

1. $N_1 \cup \dots \cup N_i = M$
2. für alle $N_i, N_j \in X$, entweder $N_i = N_j$ oder $N_i \cap N_j = \emptyset$.

3.4 Einige Beobachtungen

Wir haben bereits gesagt, dass Wahrscheinlichkeiten Zahlen in $[0, 1]$ sind, wobei wir die Korrespondenz haben

$$\begin{aligned} 0 &\cong \text{Unmöglichkeit} \\ 1 &\cong \text{Sicherheit} \end{aligned}$$

Nun haben wir, aus logischen Gründen folgendes:

$$(2) \quad P(A) \leq P(A \text{ oder } B) \text{ und } P(B) \leq P(A \text{ oder } B)$$

(In Zukunft schreiben wir: $P(A \cup B)$). Das ist klar: wann immer A eintritt, tritt auch A oder B ein, also ist das Ereignis wahrscheinlicher etc. Ebenso klar ist:

$$(3) \quad P(A \text{ und } B) \leq P(A) \text{ und } P(A \text{ und } B) \leq P(B)$$

(In Zukunft schreiben wir: $P(A \cap B)$ oder einfach $P(AB)$). Das ist klar: die Wahrscheinlichkeit, dass sie bei Ihrer nächsten Radfahrt angefahren werden ist größer als die, dass sie angefahren werden und im Zuge dessen 50euro finden.

Gleichzeitig haben wir folgendes: sei \perp ein Ereignis, das vollkommen unmöglich ist, z.B. Sie würfeln (mit einem handelsüblichen Würfel) eine 7. Dann haben wir natürlich:

$$(4) \quad P(A \cap \perp) = 0; P(A \cup \perp) = P(A)$$

Also, in Worten: \perp ist *absorbierend für Konjunktion* und *neutral für Disjunktion*.

Umgekehrt, sei \top ein Ereignis, dessen Eintritt sicher ist, z.B. dass Sie eine Zahl zwischen 1 und 6 würfeln. Dann haben wir

$$(5) \quad P(A \cap \top) = P(A); P(A \cup \top) = 1$$

Also gilt: \top ist absorbierend für Disjunktion, und neutral für Konjunktion. Nun haben wir, nach Annahme:

$$(6) \quad P(\top) = 1; P(\perp) = 0$$

Wir suchen also Operationen, für die 1, 0 jeweils neutral bzw. absorbierend sind. Das wird erfüllt von den Operationen $+$ und \cdot :

$$(7) \quad n + 0 = n \quad n \cdot 0 = 0$$

Ebenso haben wir:

$$(8) \quad n \cdot m \leq n \quad \text{und} \quad n \cdot m \leq m \quad , \text{ für } n, m \in [0, 1],$$

sowie:

$$(9) \quad n \cdot 1 = n \quad n + 1 \geq 1$$

sowie:

$$(10) \quad n \leq n + m \text{ und } m \leq n + m, \text{ für } n, m \in [0, 1]$$

Wir haben also folgende Korrespondenz:

$$\begin{aligned} \text{Konjunktion} &\cong \cdot \\ \text{Disjunktion} &\cong + \end{aligned}$$

Das Problem ist, dass sich in dem einfachen Fall die Wahrscheinlichkeiten *nicht* auf 1 aufsummieren. Wir haben eine Korrespondenz, aber das ist noch zu einfach gedacht. Das sieht man auch an folgendem Beispiel:

$$(11) \quad P(A \cap A) = P(A) \neq P(A) \cdot P(A)$$

sowie

$$(12) \quad P(A \cup A) = P(A) \neq P(A) + P(A)$$

Konjunktion und Disjunktion sind also **idempotent**, im Gegensatz zur Addition und Multiplikation. Die Materie ist also durchaus komplex; es gibt allerdings eine wunderbar elegante Lösung, die uns mit allen nötigen Rechenregeln versorgt.

3.5 Definition von Wahrscheinlichkeitsräumen

Folgende Definition stammt von Kolmogorov, und ist das Ergebnis langer Überlegungen und Dispute.

Definition 2 *Ein Wahrscheinlichkeitsraum ist ein Tripel $(\Omega, \mathfrak{A}, P)$, wobei $\mathfrak{A} \subseteq \wp(\Omega)$ eine Boolesche Algebra ist, und $P : \mathfrak{A} \rightarrow [0, 1]$ eine Wahrscheinlichkeitsfunktion, so dass*

1. $P(\Omega) = 1$;
2. $P(\emptyset) = 0$, und
3. falls A_1, A_2, \dots, A_n paarweise disjunkt sind, dann ist

$$P(\bigcup_{i=1}^n A_i) = \sum_{i=1}^n P(A_i)$$

Zur Erklärung: mit **paarweise disjunkt** meint man: für alle i, j so dass $1 \leq i, j \leq n$, falls $i \neq j$, dann ist $A_i \cap A_j = \emptyset$.

Die Bedingung der Booleschen Algebra ist wie folgt zu verstehen: falls $A, B \subseteq \Omega$ Ereignisse sind, die eine Wahrscheinlichkeit haben, dann haben auch die Ereignisse $A \cup B$ (d.h.: A oder B trifft ein), $A \cap B$ (d.h. beide A und B treffen ein) und \bar{A} (d.h. A trifft nicht ein) eine Wahrscheinlichkeit.

3.6 Ereignisse und Ergebnisse

Wir nennen eine Menge $A \subseteq \Omega$ ein **Ereignis**; wir nennen $a \in \Omega$ ein **Ergebnis**. Meistens entspricht ein Ergebnis a einem Ereignis $\{a\}$. Aber nicht immer ist das intuitiv: nehmen wir an, wir würfeln mit zwei Würfeln, wobei unsere Ergebnisse die Form haben

$$\langle m, n \rangle$$

Nun ist “der erste Wurf ist eine 2” kein Ergebnis, sondern ein Ereignis, nämlich das Ereignis

$$\{\langle 2, 1 \rangle, \dots, \langle 2, 6 \rangle\}$$

Daher weisen wir Wahrscheinlichkeiten normalerweise Ereignissen zu, nicht Ergebnissen.

3.7 Die Komplement-Regel

Wir kommen nun zu den Rechenregeln. Die Regel für die Berechnung des Komplementes $P(\bar{A})$ aus $P(A)$ lautet wie folgt:

$$(10) \quad P(\bar{A}) = 1 - P(A)$$

Das lässt sich sehr einfach ableiten: wir haben

1. $P(A \cup \bar{A}) = P(\Omega) = 1$ und
2. $A \cap \bar{A} = \emptyset$;

also:

$$\begin{aligned} 1 &= P(A \cup \bar{A}) \\ &= P(A) + P(\bar{A}) \\ \Leftrightarrow 1 - P(A) &= P(\bar{A}) \end{aligned}$$

3.8 Die Summenregel

Die Summenregel erlaubt es uns, die logische Disjunktion rechnerisch aufzulösen. Die Summenregel lautet:

$$(11) \quad P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Intuitiv bedeutet das: um die Wahrscheinlichkeit einer Disjunktion zu berechnen, reicht es die Wahrscheinlichkeiten zu addieren, wenn man nur die Wahrscheinlichkeitsmasse abzieht, die auf die Konjunktion beider Ereignisse entfällt (illustrierbar mittels Venn-Diagramm). Das lässt sich wie folgt ableiten aus den Axiomen:

$$\begin{aligned}
 P(A \cup B) &= P(A \cup (B \cap \bar{A})) && \text{(Mengenlehre)} \\
 &= P(A) + P(B \cap \bar{A}) && \text{(Disjunkte Mengen)} \\
 &= P(A) + P(B \cap (\bar{A} \cup \bar{B})) && \text{(Mengenlehre)} \\
 &= P(A) + P(B \cap (\overline{A \cap B})) && \text{(Mengenlehre)} \\
 &= P(A) + P(\overline{B \cup (A \cap B)}) && \text{(Mengenlehre)} \\
 &= P(A) + P(\overline{B \cup (A \cap B)}) && \text{(Mengenlehre)} \\
 &= P(A) + (1 - P(\overline{B \cup (A \cap B)})) && \text{(Subtraktionsregel)} \\
 &= P(A) + (1 - P(\overline{B}) + P(A \cap B)) && \text{(Disjunkte Mengen)} \\
 &= P(A) + (1 - (1 - P(B)) + P(A \cap B)) && \text{(Disjunkte Mengen)} \\
 &= P(A) + (1 - (1 - P(B))) - P(A \cap B) && \text{(Arithmetik)} \\
 &= P(A) + (1 - 1) + P(B) - P(A \cap B) && \text{(Arithmetik)} \\
 &= P(A) + P(B) - P(A \cap B) && \text{(Arithmetik)}
 \end{aligned}$$

3.9 Die Produktregel

Um die Konjunktion sinnvoll zu interpretieren, brauchen wir die Definition der *bedingten Wahrscheinlichkeit*. Wir definieren

$$(12) \quad P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Nehmen Sie nun an wir suchen die Wahrscheinlichkeit $P(A \cap B)$; wir bekommen sie durch eine ganz simple Termumformung:

$$(13) \quad P(A \cap B) = P(A|B)P(B)$$

Da $P(A \cap B) = P(B \cap A)$ (\cap ist kommutativ), bekommen wir also die Produktregel:

$$(14) \quad P(A \cap B) = P(A|B)P(B) = P(B|A)P(A)$$

Intuitiv ist das wie folgt zu verstehen: wenn A und B eintreffen, dann bedeutet das 1. A eintritt und 2. unter dieser Voraussetzung B eintritt (oder umgekehrt). Wichtig ist: $P(A|B)$ sagt nichts über zeitliche Reihenfolge! So ist die Formel intuitiv richtig. Wir werden später noch mehr zum Konzept der bedingten Wahrscheinlichkeit erfahren.

Diese Umformung mag einem auf Anhieb nicht sehr hilfreich erscheinen, allerdings ist sie eines der zentralen Gesetze. Denn in der Praxis kennen wir oft bedingte Wahrscheinlichkeiten besser als unbedingte, so dass wir uns das Gesetz leicht zunutze machen können. Es gibt übrigens noch eine allgemeinere Form der Produktregel:

$$(15) \quad P(A \cap B|X) = P(A|BX)P(B|X) = P(B|AX)P(A|X)$$

Das generalisiert die letzte Formel, da X beliebig (auch leer) sein kann.

3.10 Das sog. Bayessche Gesetz und seine Bedeutung

Das Bayessche Gesetz bzw. Theorem ist im Prinzip auch nichts anderes als eine Term-Umformung, vorausgesetzt alle Definitionen soweit. Es sieht wie folgt aus:

$$(16) \quad P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A \cap B) P(A)}{P(B) P(A)} = \frac{P(B \cap A) P(A)}{P(A) P(B)} = P(B|A) \frac{P(A)}{P(B)}$$

Die Bedeutung ist folgende: wir haben Wahrscheinlichkeitstheorie eingeführt als ein Werkzeug, um uns rational zu verhalten. Noch häufiger werden wir sie benutzen, um eine rationale Sicht der Dinge zu bekommen. Die Frage wird sein: gegeben unsere Beobachtungen, was ist die wahrscheinlichste Annahme über die Natur der Dinge? Seien also B unsere Beobachtungen, H eine Hypothese (über zugrundeliegende Wahrscheinlichkeiten); wir suchen also $P(H|B)$. Das lässt sich aber nicht ohne weiteres errechnen; wir bekommen normalerweise nur $P(B|H)$! Das Bayessche Gesetz erlaubt uns aber, von $P(B|H)$ zu $P(H|B)$ zu gelangen – mit ein paar Seitenannahmen, doch dazu später mehr.

4 Exkurs: Conditional Random Fields

4.1 Definition

Wir geben hier eine Einführung in sog. **conditional random fields**, nicht weil sie im engeren Sinne mit DL zu tun haben, sondern weil die Praxis gezeigt hat, dass für manche Anwendungen (sequence to sequence) die beste Performance durch die Kombination von tiefer Architektur und CRF erreicht wird.

CRF basieren auf ungerichteten Graphen, also Strukturen (V, E) , wobei V eine Menge ist, $E \subseteq V^2$ die Menge der Kanten zwischen den Knoten. Wir weisen, so wie immer in probabilistischen graphischen Modellen, jedem Knoten $v \in V$ eine Zufallsvariable zu; wir haben also eine Familie $\mathbf{Y} = Y_v : v \in V$. Die wichtigste graphentheoretische Relation ist die **Nachbarschaft** im Graph, $v \sim v'$ gdw. $(v, v') \in E$. Da unser Graph ungerichtet ist, ist \sim symmetrisch.

Was man zusätzlich braucht, ist eine Menge von **Beobachtungslabeln**, die wir X nennen, die label selbst nennen wir $x \in X$. Der Sinn hierhinter ist folgender: CRF transduzieren Sequenzen zu Sequenzen; X sind die möglichen Beobachtungen; die Eingabe befindet sich also in X^* (endliche Folge über X). Die Ausgabe sind die Werte, welche die Zufallsvariablen annehmen (schreiben wir y). Die Beobachtungen X sind *keine Werte* der Zufallsvariablen!

Der nächste Schritt besteht darin, dass wir eine Menge von **konditionalen Wahrscheinlichkeitsverteilungen** definieren. Hier zählen aber nicht nur die Werte, welche die anderen Variablen annehmen, sondern auch die Beobachtung, die wir in der Variable machen.

$$(17) \quad P(Y_v|x, Y_{v'} : v \sim v')$$

Wichtig ist hierbei, dass gilt: jede Verteilung ist völlig durch 1. die Beobachtung und 2. seine Nachbarverteilungen bestimmt. Das definiert man mittels folgender Gleichung:

$$(18) \quad P(Y_v|x, Y_{v'} : v \sim v', x) = P(Y_v|x, Y_{v'} : v \neq v')$$

Wir haben im Graphen also die **Markov-Eigenschaft**: nur die Nachbarn bestimmen einander. Das bedeutet aber nicht, dass zwei nicht benachbarte Knoten sich nicht beeinflussen: sie beeinflussen ihre Nachbarn, die wiederum die Nachbarn beeinflussen etc. Man stellt sich den Graphen also am besten

als ein Netz vor, durch welches Information fließt, und in welchem der Fluss dadurch **blockiert** wird, dass wir den Wert einer Variable kennen.

Was wir definieren wollen ist normalerweise folgendes: eine Wahrscheinlichkeit

$$(19) \quad P(\vec{y}|\vec{x})$$

die uns sagt, wie wahrscheinlich ein Labelling für eine Eingabesequenz ist. Am Ende suchen wir natürlich die wahrscheinlichste Ausgabe, aber das ist ein anderes Problem, das man **Dekodierung** nennt. Wahrscheinlichkeiten berechnen wir wie folgt: wir haben sog. Merkmalsfunktionen f_j mit Gewichten λ_j . Wir definieren dann

$$(20) \quad score(Ys = y|x) = \sum_{j=1}^m \sum_{i=1}^n \lambda_j f_j(x, x_{j:i \sim j})$$

???Hier die Frage: was spielt eine Rolle? Natürlich die Beobachtung selbst; aber dann: die umgebenden label, oder wie sie gelabelt werden, oder beides? Potentiell beides. Auf Sequenzen erweitert bedeutet das:

$$(21) \quad score(Y_i = y|s_{in}) = \sum_{j=1}^m \lambda_j f_j(s_{in}, y, Y_j = y_j : i \sim j)$$

Wir haben hier also das label der i -ten Stelle, das sich als score errechnet aus Eingabesatz (alle Beobachtungen stehen zur Verfügung!), sowie den labeln die benachbarten Variablen. Der score der Ausgabesequenz errechnet sich wie folgt:

$$(22) \quad score(s_{out}|s_{in}) = \sum_{j=1}^m \sum_{i=1}^n \lambda_j f_j(s_{in}, y, Y_j = y_j : i \sim j)$$

Daraus wird eine Wahrscheinlichkeit (was nicht unbedingt nötig ist, aber pro forma gemacht wird) mittels

$$(23) \quad P(s_{out}|s_{in}) = \frac{\exp(score(s_{out}|s_{in}))}{\sum_{s'_{out}} \exp(score(s'_{out}|s_{in}))}$$

Der Index i läuft über die Sequenz; wenn wir also die triviale Sequenz nehmen (Länge 1), dann bekommen wir:

$$(24) \quad \text{score}(Y_1 = y_1 | x_1) = \sum_{j=1}^m \lambda_j f_j(y_1, x_1, y_1)$$

Das Modell ist sehr liberal und muss für jede praktische Anwendung deutlich restringiert werden. Insbesondere die Merkmalsfunktionen müssen *von Hand* ausgearbeitet werden – und das ist immer schlecht!

4.2 Ein Beispiel

Ein einfaches und geläufiges Beispiel ist das bekannte Problem des POS-tagging: gegeben einen Satz (d.i. eine Folge von Wörtern) sollen wir eine ebenso lange Folge von POS-tags generieren. In diesem Fall wählen wir zunächst eine “Fenstergröße”, die uns angibt, in welchem Rahmen wir die zugewiesenen POS-tags voneinander abhängig machen wollen, z.B. 2 nach vorne, 2 nach hinten. Das bedeutet, dass wir einen Graphen mit 5 Knoten ????

4.3 Parameter schätzen

5 Lineare Modelle

5.1 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 $f : \mathbb{R} \rightarrow \mathbb{R}$, die die Form haben soll:

$$(25) \quad f(x) = ax + b$$

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

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

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

$$(26) \quad \operatorname{argmin}_{a,b \in \mathbb{R}} \sum_{(x,y) \in D} (ax + b - y)^2$$

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 $|\cdot|$ 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 berücksichtigen müssen. Daher ist die Funktion, die wir minimieren müssen, eigentlich folgende:

$$(27) \quad \sum_{(a,b) \in D} (ax + y - b)^2 = (a_1x + y - b_1)^2 + \dots + (a_nx + y - b_n)^2$$

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

$$(28) \quad (a_1x + y - b_1)^2 = a_1^2x^2 + y^2 + a_1xy - a_1b_1x - b_1y + b_1^2$$

Diese Umformung machen wir für alle Summanden, und da alle dieselben Variablenformen aufweisen, können wir aufaddieren; bekommen also:

$$(29) \quad \sum_{(a,b) \in D} (ax + y - b)^2 = ax^2 + by^2 + cxy + dx + ey + f$$

Wobei $a = a_1^2 + \dots + a_n^2$, $b = b_1^2 + \dots + b_n^2$. Wir suchen nun einfach

$$(30) \quad \operatorname{argmin}_{x,y \in \mathbb{R}} ax^2 + by^2 + cxy + dx + ey + f$$

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

$$(31) \quad \nabla f(x, y) = ((2ax + cy + d), (2by + cx + e))$$

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

$$(32) \quad 2ax + cy + d = 0$$

$$(33) \quad 2by + cx + e = 0$$

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

$$(34) \quad x = -\frac{cy + d}{2a}$$

also

$$(35) \quad 2by - \frac{c^2y + cd}{2a} + e = 0 \leftrightarrow$$

$$(36) \quad y = \frac{cd - 2ae}{4ab - c^2}$$

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

5.2 Der komplexe lineare Fall

Im komplexeren lineare Fall nehmen wir an, dass

$$(37) \quad f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$$

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, \dots, 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.

$$(38) \quad f(x_0, \dots, x_n) = \sum_{(a,b) \in D} (a^n x_n + a^{n-1} x_{n-1} + \dots + x_0 - b)^2$$

Hier sind a^n etc. und b fixe reelle Zahlen, während die Variablen nur linear auftreten. Sprich:

Polynomiale Regression ist lineare Regression auf einem speziellen höherdimensionalen Datensatz der Form (a^n, a^{n-1}, \dots, a)

Wir müssen nun

$$(39) \quad \nabla f : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^{n+1}$$

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 $f : \mathbb{R} \rightarrow \mathbb{R}$ suchen wir eine Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$, die die Form hat

$$(40) \quad f(x_1, \dots, x_n) = a_1 x_1 + a_2 x_2 + \dots + a_n x_n + b$$

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

5.3 Lineare Regression in R

In R gibt es ein einfaches Kommando zur (einfachen) linearen 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:

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

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

$$(41) \quad f(x) = ax + b$$

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:

$$(42) \quad lm1(x) = -0.7896x + 3.4308$$

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

$$(43) \text{ residuals}(f) = (f(x_1) - y_1, \dots, f(x_n) - y_n)$$

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?

5.4 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

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

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 vergrößert sich das Ergebnis mit einer Vergrößerung 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-Komponenten *monoton fallend*. Anders gesagt: ob XOR in der ersten/zweiten Komponente 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

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

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

$$(44) \quad \phi(x, y) = (x \cdot y, x + y)$$

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!

5.5 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

$$(45) \quad f(x) = ax + b$$

$$(46) \quad g(x) = cx + d$$

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 entscheidend – 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

$$(47) \quad l_1(x)$$

$$(48) \quad n_1 \circ l_1(x)$$

$$(49) \quad l_2 \circ n_1 \circ l_1(x)$$

$$(50) \quad n_2 \circ l_2 \circ n_1 \circ l_1(x)$$

$$(51) \quad \dots$$

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.

5.6 Höhere Dimensionalität

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

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

aufgefasst. Wir werden jetzt die allgemeinere Perspektive nehmen mit

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

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

$$(52) \quad L(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{a}$$

wobei \mathbf{M} eine Matrix ist (Dimension $m \times n$) und \mathbf{a} ein Vektor (Dimension m). 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

$$(53) \quad \mathbf{M} = \begin{pmatrix} 2.3 & 1.2 & 4 \\ 1 & 3.7 & 10 \end{pmatrix}$$

$$(54) \quad \mathbf{a} = (5, 6.3)$$

Dann haben wir also:

$$(55) \quad L(\mathbf{x}) = (2.3x_1 + 1.2x_2 + 4x_3, 1x_1 + 3.7x_2 + 10x_3) + (5, 6.3)$$

$$(56) \quad = (2.3x_1 + 1.2x_2 + 4x_3 + 5, 1x_1 + 3.7x_2 + 10x_3 + 6.3)$$

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.

Wie funktioniert nun höherdimensionale Regression? In diesem Fall haben wir einen Datensatz $D \subseteq \mathbb{R}^{n+m}$, wobei wir schreiben

$$D = \{(\vec{v}_1, \vec{w}_1), \dots, (\vec{v}_i, \vec{w}_i)\}$$

Sprich, wir haben Vektoren als Ein- und Ausgaben. Wir suchen also nun eine ganze Reihe Parameter; genauer gesagt suchen wir eine Matrix $M \in \mathbb{R}^{m \times n}$, $\vec{b} \in \mathbb{R}^m$, so dass wir

$$(57) \quad \operatorname{argmin}_{M, \vec{b}} \sum_{(\vec{v}, \vec{w}) \in D} d(M\vec{v} + \vec{b})$$

Hier brauchen wir, als Kostenfunktion, die Metrik $d(\cdot, \cdot)$, normalerweise die euklidische Distanz. Das bedeutet wir bekommen:

$$(58) \quad \operatorname{argmin}_{M, \vec{b}} \sum_{(\vec{v}, \vec{w}) \in D} \|(M\vec{v} - \vec{b})\|_2$$

Wie kann man diese Funktion minimieren? Die überraschende Antwort lautet: die Regression einer Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ zerfällt in m separate lineare Regressionen:

- Wir erhalten M_1 , und \mathbf{b}_1 (also erste Zeile von M , erste Komponente von \vec{b}) als eine einfache lineare Regression über $D_1 = \{(\vec{v}_1, (\vec{w}_1)_1), \dots, (\vec{v}_i, (\vec{w}_i)_1)\}$
- ...
- Wir erhalten M_m , und \mathbf{b}_m (also erste Zeile von M , erste Komponente von \vec{b}) als eine einfache lineare Regression über $D_1 = \{(\vec{v}_1, (\vec{w}_1)_m), \dots, (\vec{v}_i, (\vec{w}_i)_m)\}$

Warum ist das so? Weil alle Operationen, bis hin zur finalen Summierung in $\|\vec{x}\|$, streng unabhängig sind!

Man beachte aber dass die Metrik dennoch eine Rolle spielt; die Metrik bestimmt letztendlich, was stärker gewichtet wird: einzelne Ausreisser oder

viele kleinere Abweichungen (je größer p , desto stärker werden Ausreißer bestraft).

Auch diese linearen Modelle behalten die zentralen 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

$$(59) \quad L_1 \circ L_2(\mathbf{x}) = L_1(L_2(\mathbf{x}))$$

2. Wenn wir Gleichung 55 betrachten, dann können wir mit elementaren Methoden die optimale Lösung finden. Das bedeutet: das Optimierungsproblem ist lösbar, wir finden immer die beste Lösung.

6 Die Logistische Regression und das einfache Perzeptron

6.1 Das einfache Perzeptron

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

$$(60) \quad S(x) = \frac{1}{1 + e^{-x}}$$

Einige Beobachtungen: wir haben

- $\lim_{x \rightarrow -\infty} S(x) = 0$
- $\lim_{x \rightarrow \infty} S(x) = 1$
- $S(x) \in (0, 1)$ 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=c(0,1))
```

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

$$(61) \quad T(x) = \frac{x}{1 + |x|}$$

$$(62) \quad U(x) = \frac{x}{1 + x^2}$$

$$(63) \quad \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 - \frac{2}{e^{2x} + 1}$$

Was allen diesen Funktionen gemeinsam ist, ist dass sie im Bereich um 0 relativ schnell von 0 bzw. -1 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 $\mathbb{R} \rightarrow \mathbb{R}$ 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:

$$(64) \quad P(X = A|x) = S(f(x))$$

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 $[0, 1]$:

$$f : \mathbb{R} \rightarrow \mathbb{R} \implies S \circ f : \mathbb{R} \rightarrow [0, 1]$$

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

$$(65) \quad C(x) = \begin{cases} A, & \text{falls } P(X = A|x) > 0.5 \\ B & \text{andernfalls} \end{cases}$$

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 $\mathbf{x} \in \mathbb{R}^n$.

6.2 Bedeutung

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

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

Wir tun das aber mittels zweier Zwischenschritte:

$$\mathbb{R}^n \xrightarrow{f} \mathbb{R} \xrightarrow{s} \{0, 1\}$$

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. Man braucht dann

- Einen Schwellenwert s , an dem die Klassifikation umspringt
- und erhält so einen Klassifikator C .

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 einen Klassifikator

$$C : \mathbb{R} \rightarrow \{0, 1\}$$

Wir wissen, dass

- Eine lineare Funktion hat eine Gerade als Graphen, kann also einen gewissen Wert nur einmal annehmen
- Ein Polynom zweiten Grades hat eine Maximalstelle, kann also einen gewissen Wert höchstens zweimal annehmen.
- ...

- Ein Polynom n ten Grades hat $n-1$ Maximalstellen, $n-2$ Wendepunkte, kann also einen gewissen Wert höchstens $n-1$ -mal annehmen.

Dann heißt das soviel wie:

- Falls wir eine zugrundeliegende lineare Funktion haben, dann gibt es ein $\alpha \in \mathbb{R}$, so dass falls $x < \alpha$, dann $C(x) = 1$ (bzw. 0), ansonsten $C(x) = 0$ (bzw. 1). Wir spalten also den Eingaberaum \mathbb{R} in zwei Teile.
- Falls wir ein zugrundeliegendes Polynom zweiten Grades f_2 haben, dann gibt $\alpha_1, \alpha_2 \in \mathbb{R}$, so dass falls $\alpha_1 < x < \alpha_2$, dann $C(x) = 1$ (bzw. 0), ansonsten $C(x) = 0$ (bzw. 1). Wir spalten also den Eingaberaum \mathbb{R} in drei Teile.
- ...
- Falls wir ein Polynom n -ten Grades f_n haben, haben wir $\alpha_1, \dots, \alpha_n$, die \mathbb{R} in $n+1$ Teile spalten, und wir klassifizieren auf dieser Basis.

Dasselbe generalisiert sich in höhere Dimensionen, wir nur weniger anschaulich. Allgemeiner kann man sagen:

Sei C ein Klassifikator basierend auf einem einfachen Perzeptron $S \circ f$, S eine Aktivierungsfunktion, $f : \mathbb{R}^n \rightarrow \mathbb{R}$ eine lineare Funktion, und $C(\vec{x}) = 1$ gdw. $S(f(\vec{x})) > s$.

Dann gibt es eine Hyperebene H im n -dimensionalen Raum, so dass gilt:

$$C(\vec{x}) = 1 \text{ gdw. } \vec{x} \text{ oberhalb/unterhalb der Hyperebene liegt.}$$

Zur Erinnerung:

- eine Hyperebene in \mathbb{R} ist ein Punkt
- eine Hyperebene in \mathbb{R}^2 ist eine Gerade
- eine Hyperebene in \mathbb{R}^3 ist eine Ebene
- ...

Wie kann man dieses Konzept formal beschreiben in aller Allgemeinheit?
Eine Hyperebene im $n + 1$ -dimensionalen Raum ist nach Definition eine Menge von Punkten

$$(66) \quad H = \{\vec{s} + \lambda_1 \vec{r}_1 + \dots + \lambda_{n-1} \vec{r}_n : \lambda_1, \dots, \lambda_n \in \mathbb{R}\}$$

Man kann das auch äquivalent als Gleichung darstellen: ein Hyperebene ist die Menge der Lösungen einer Gleichung, also

$$(67) \quad H = \{(x_1, \dots, x_n) \in \mathbb{R}^n : a_1 x_1 + \dots + a_n x_n + a_{n+1} = 0\}$$

wobei die Parameter a_1, \dots, a_{n+1} aus den $\lambda_1, \dots, \lambda_n$ errechnet werden können.
Man sagt nun: $\vec{y} = (y_1, \dots, y_n)$ liegt **oberhalb** von H , falls

$$(68) \quad a_1 y_1 + \dots + a_n y_n + a_{n+1} = 0$$

Dieses Konzept lässt sich nun wieder generalisieren zu Polynomen beliebigen Grades; das wird aber schnell unanschaulich.

6.3 Exkurs: Lernen & Anwendung von einfachen Perzeptronen

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):

$$(69) \quad f(x) = 0.1786x - 0.6786$$

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 definieren:

$$(70) \quad P(\text{bestehen} | n \text{ Stunden lernen}) = \frac{1}{1 + e^{-f(x)}} = \frac{1}{1 + e^{-(0.1786x - 0.6786)}}$$

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:

$$(71) \quad \operatorname{argmin}_{a,b \in \mathbb{R}} \sum_{(x,y) \in D} \frac{1}{1 + e^{-(ax+b)}} - y$$

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 71 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**:

$$(72) \quad \frac{(\theta_{ML} - \theta)^2}{\text{var}(\theta_{ML})}$$

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:

$$(73) \quad R(\omega) := \frac{P(\omega|H_0)}{P(\omega|H_1)} \quad (\text{Sonderfall für } P(\omega|H_1) = 0)$$

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:

$$(74) \quad S_t(\omega) = \begin{cases} H_0, & \text{falls } R(\omega) > t \\ H_1 & \text{andernfalls.} \end{cases}$$

Das können wir/Sie nun ausrechnen:

6.4 Höherdimensionale Modelle mit nichtlinearen Funktionen

Das Modell des einfachen Perzeptrons wird geradeaus erweitert auf unseren Fall: wir bekommen

$$(75) \quad per(\vec{x}) = n(M\vec{x} + \vec{b}) = (n((M\vec{x} + \vec{b})_1), \dots, n((M\vec{x} + \vec{b})_m))$$

Wir haben bereits besprochen dass ein einfaches (eindimensionales) Perzeptron im Prinzip eine Art (weicher) Hyperebene erzeugt: wir haben oberhalb der Hyperebene den Wert (beinahe) 1, unterhalb den Wert (beinahe) 0; während um die Ebene selbst ein kurzes Moment der Ungewissheit steht.

Was also macht ein multidimensionales Perzeptron? Wie wir gesehen haben ergibt sich eine Funktion

$$(76) \quad f(\vec{x}) = M\vec{x} + \vec{b}$$

als eine unabhängige Applikation verschiedener einfacher linearer Funktionen. Das bedeutet: $per(\vec{x})$ ist wiederum nur eine Liste von unabhängigen Perzeptronen, und macht nichts anderes als m solche Hyperebenen durch den Raum zu ziehen.

Das bedeutet: ein höherdimensionales Perzeptron $per : \mathbb{R}^n \rightarrow \mathbb{R}^m$ unterteilt den n -dimensionalen Raum mit m Hyperebenen, und jede (binarisierte) Ausgabe steht für einen der 2^m Quadranten im n -dimensionalen Raum. NB: es spielt hierbei keine Rolle ob $m < n$, $m = n$ oder $n < m$; das ist grundsätzlich unerheblich.

Wir können uns das Perzeptron also geometrisch als Fallunterscheidung oder Quadranten im Raum vorstellen.

6.5 Exkurs: nearest neighbour Regression

Nearest neighbour regression ist eine denkbar simple Methode: wir haben eine Funktion $f : V \rightarrow C$, 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

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

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 Skalarmultiplikation)
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:

$$(77) \quad \|(v_1, \dots, v_n)\| = \sqrt{v_1^2 + \dots + v_n^2}$$

In dieser geometrischen Interpretation wird Bedingung 3 zur **Dreiecksungleichung**: in jedem rechteckigen Dreieck ist die Länge der Hypotenuse 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:

$$(78) \quad \|(v_1, \dots, v_n)\|_p = \left(\sum_{i=1}^n |v_i|^p \right)^{\frac{1}{p}}$$

Für $p = 1$ vereinfacht sich das zu

$$(79) \quad \|(v_1, \dots, v_n)\|_1 = \sum_{i=1}^n |v_i|$$

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

$$(80) \quad d(\vec{v}, \vec{w}) = \|\vec{v} - \vec{w}\|$$

wobei natürlich

$$(81) \quad \vec{v} - \vec{w} = (v_1 - w_1, \dots, v_i - w_i)$$

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

$$(82) \quad d_2(\vec{v}, \vec{w}) = \|\vec{v} - \vec{w}\|_2$$

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

$$(83) \quad nn_D(\vec{v}) = \operatorname{argmin}_{(\vec{w}, c) \in D} d(\vec{v}, \vec{w})$$

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 $D : V \rightarrow C$ auf)

$$(84) \quad NNR_D(\vec{v}) = D(nn_D(\vec{v}))$$

In Wort: $NNR_D : V \rightarrow C$ ist die nearest neighbour regression über D , und diese (vollständige) Funktion funktioniert, indem sie für jeden Eingabevektor \vec{v} 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

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

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.

$$(\text{Abstraktes}) \text{ ML-Modell} \xrightarrow[\text{Daten}]{} (\text{Konkrete}) \text{ Funktion} \xrightarrow[\text{Eingabe}]{} \text{Ausgabe}$$

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.

7 Nichtlineare Funktionen: Aktivierung u.a.

7.1 Überblick

Bei tiefen Architekturen gibt es zwei Arten von nichtlinearen Funktionen zu unterscheiden:

1. Innere Funktionen
2. Äußere Funktionen

Der Unterschied betrifft nicht primär die Art der Funktion, sondern die Art der Anwendung:

- Innere Funktionen werden appliziert in inneren Layern des MLP
- Äußere Funktionen werden appliziert als Ausgabe der Funktion

Hieraus ergibt sich folgendes: Äußere Funktionen unterliegen dem Zwang, dass die Ausgabe, im Rahmen der Aufgabe, die das Netz erfüllen soll, *interpretierbar* sein muss.

Beispiel 3 *Z.B. nimm an, wir haben eine Bilderkennung über 50 Kategorien. Dann sollte uns die Ausgabe Funktion also eine Wahrscheinlichkeitsverteilung über die 50 Kategorien liefern, üblicherweise kodiert als Vektor $\vec{w} \in \mathbb{R}^{50}$, wobei $\sum \vec{w} = 1$.*

Innere Funktionen hingegen haben keinerlei derartige Beschränkungen; sie geben einfach Werte weiter innerhalb des Netzes. Allgemein gilt:

- Innere Funktionen haben keinen Einfluss auf die Dimensionalität.
- Innere Funktionen geben Parameter der Komponenten unabhängig voneinander weiter, d.h. $S(x_1, \dots, x_n) = (S(x_1), \dots, S(x_n))$.

Gerade letzteres gilt für äußere Funktionen üblicherweise nicht!

In tiefen Architekturen gilt: 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

$$M(\vec{x}) \in \mathbb{R} \text{ (} M \text{ steht hier für Modell)}$$

(z.B. bei binärer Klassifikation). In diesem Fall suchen wir dann eine Aktivierungsfunktion

$$(85) \quad A : \mathbb{R}^n \rightarrow \mathbb{R}$$

7.2 Innere Funktionen (Aktivierungsfunktionen)

NB: diese Funktionen sind meistens Funktionen $f : \mathbb{R} \rightarrow \mathbb{R}$. 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**

$$(86) \quad S(x) = \frac{1}{1 + e^{-x}}$$

Einige Beobachtungen: wir haben

- $S(0.5) = 0$
- $\lim_{x \rightarrow -\infty} S(x) = 0$
- $\lim_{x \rightarrow \infty} S(x) = 1$
- $S(x) \in (0, 1)$ 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

$$(87) \quad T(x) = \frac{x}{1 + |x|}$$

$$(88) \quad U(x) = \frac{x}{1 + x^2}$$

$$(89) \quad \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 - \frac{2}{e^{2x} + 1}$$

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 $\mathbb{R} \rightarrow \mathbb{R}$ definiert sind. Aktivierungsfunktionen spielen in neuronalen Netzen eine große Rolle, und werden normalerweise auf die einzelnen Komponenten von Vektoren angewendet.

7.3 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

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

Wir können das erweitern auf beliebige

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

mittels der Gleichung

$$(90) \quad f(x_1, \dots, x_m) = (f(x_1), \dots, f(x_m))$$

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

7.4 Rectified linear units und ihre Generalisierung

ReLU sind definiert durch:

$$(91) \quad g(x) = \max\{0, x\}$$

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:

$$(92) \quad h(\mathbf{x}) = g(M\mathbf{x} + \mathbf{b})$$

Das gute an diesen Funktionen ist natürlich ihre Einfachheit. Das Problem ist: 1. Sie ist nicht differenzierbar an einem Punkt. Das ist aber für heuristische Optimierung nicht so schlimm. Problematischer: sie haben ein langes Plateau, das ist wirklich ein Problem für Optimierung!

Es gibt daher 2 Generalisierungen von ReLUs:

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

$$(93) \quad g(\vec{x}, \alpha)_i = \max(0, x_i) + \alpha_i \min(0, x_i)$$

Im Fall von $\alpha_i = 0$ haben wir eine ReLU-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 10^{-n} setzen, mit z.B. $n = 100$; das ergibt **leaky** ReLU. In **parametrischen** ReLU 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, \dots, G_k , wobei also

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

jeweils Gruppen von k (beliebig, aber fest) gewählten Komponenten aus den n Eingabekomponenten sind.

Die Ausgabe der maxout Funktion ist wiederum ein Vektor, wobei

$$(94) \quad g : \mathbb{R}^n \rightarrow \mathbb{R}^j$$

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

$$(95) \quad g(\vec{z})_i = \max(z_j : j \in G_i)$$

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

7.5 Ausgabefunktionen

max als (schlechter) Klassifikator Eine sehr wichtige Funktion ist die *max*-Funktion, die einfach das Maximum ausgibt:

$$(96) \quad \max(x_1, \dots, x_m) = \max\{x_1, \dots, x_m\}$$

Sie liefert also den höchsten Wert. Darauf basiert eine Art *kompmax*-Funktion, die wie folgt definiert ist:

$$(97) \quad \text{kompmax}(x_1, \dots, x_m) = i : x_i = \max\{x_1, \dots, x_m\}$$

Das liefert uns einen guten Klassifikator: jeder Komponente in (x_1, \dots, x_m) kann man auffassen als Repräsentation einer Klasse, und für eine Eingabe \vec{y} ,

$$(98) \quad \text{kompmax}(N(\vec{y})) = C(\vec{y})$$

also das Klassenlabel. Das Problem dieser Funktion ist: sie ist nicht differenzierbar. Daher ist sie unbrauchbar!

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

$$(99) \quad P(y|\vec{z}) \mapsto [0, 1]$$

wobei $y \in Y$ eine endliche Menge ist, und \vec{z} die Eingabe. Dann können wir mit **Kreuzentropie** optimieren (dazu später mehr).

Das bekommen wir in neuronalen Netzen, indem wir die mehrdimensionalen Funktionen mit A komponieren (A die Ausgabe):

$$(100) \quad P(y|\mathbf{z}) = A \circ N(\vec{z}),$$

wobei

$$(101) \quad N(\vec{z}) = \vec{x}$$

der output der (komplexen) inneren Funktion ist (z.B. ein lineares Modell, oder ein MLP). Das bedeutet, wir haben eine Funktion

$$(102) \quad P : Y \times \mathbb{R}^n \rightarrow [0, 1]$$

so dass (im Idealfall)

$$(103) \sum_{y \in Y} P(y, \mathbf{x}) = 1$$

f.a. $\mathbf{x} \in \mathbb{R}^n$.

Softmax als guter Klassifikator Daher gibt es die sog. *softmax*-Funktion, die eine ähnliche Funktion übernimmt wie *max*, aber differenzierbar ist (daher *soft*, weil sie keine Knicke hat).

$$(104) \text{softmax}(x_1, \dots, x_n)_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

Diese Notation ist gewählt um die Definition übersichtlicher zu machen; wir definieren die Ausgabe also Komponentenweise. Hieraus folgt:

- Falls $\mathbf{x} \in \mathbb{R}^n$, dann ist auch $\text{softmax}(\mathbf{x}) \in \mathbb{R}^n$. Wir ändern die Dimension also nicht!
- Wir haben $\sum(\text{softmax}(\vec{x})) = 1$, sprich: Komponenten addieren sich auf 1. Wir haben also Wahrscheinlichkeitsverteilung.
- 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 Vorteil 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 (103). Aber: die Softmax-Funktion ändert nicht die Dimension, wir müssen also sicher stellen dass schon die Eingabe die richtige Dimension hat. Das macht man normalerweise mit einer linearen Funktion (Matrix).

Wichtig ist in allen diesen Funktionen, dass wir möglichst kein 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.

8 Was können Tiefe Architekturen?

8.1 Mächtigkeit ist nicht Lernbarkeit

Es gibt zwei Fragen, die durchaus unabhängig voneinander behandelt werden sollten, 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 (es gibt immer eine Funktion die “naheliegender” ist)!

Man muss also die beiden Probleme separat behandeln. Es gilt:

Frage 1 kann man relativ leicht beantworten, ist aber für das maschinelle Lernen nicht besonders wichtig.

Frage 2 hat eine zentrale Bedeutung, wir nähern uns momentan der Frage aber erst sehr langsam an. Vielleicht kommen wir am Ende dazu!

In diesem Abschnitt geht es also erstmal um **Frage 1**, 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 unbedingt 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).

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

$$(105) \quad N(\vec{x}) = g(M\vec{x} + \vec{b})$$

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

$$(106) \quad N(\vec{x}) = N_i \circ \dots \circ N_1(\vec{x}) = g_i(M_i(\dots g_1(M_1\vec{x} + \vec{b}_1)\dots) + \vec{b}_i)$$

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 non-linearen Funktionen gäbe*. Die Mächtigkeit von tiefen Netzen besteht also in der Alternation von linearen und nicht-linearen Schichten.

Man spricht in dem Zusammenhang auch von **hidden layern**:

- eine lineare Funktion hat *kein* hidden layer
- ...
- eine Funktion der Form $N(\vec{x}) = N_i \circ \dots \circ N_1(\vec{x})$ hat $i - 1$ *hidden layer*

8.2 Universelle Approximation

Wir kommen jetzt zum sog. universal approximation theorem, dass soviel besagt wie

Universelle Approximation, Annäherung Praktisch jede berechenbare Funktion können wir approximieren mit einem neuronalen Netz mit *einem hidden layer*.

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.

Zur Erklärung:

- Eine geschlossenes Intervall hat die Form $[x, y] = \{z : x \leq z \leq y\}$
- Abzählbar heißt: Kardinalität der natürlichen Zahlen.
- Eine abzählbare Vereinigung (von Mengen) ist also eine Vereinigung der Form $\bigcup_{n \in \mathbb{N}} M_n$.
- Nehmen wir eine Funktion $f : \mathbb{R} \rightarrow \mathbb{R}$. Das **Urbild** einer Menge $M \subseteq \mathbb{R}$ ist dann definiert als

$$(107) f^{-1}[M] = \{x : f(x) \in M\}$$

Definition 4 Eine Funktion f ist Borel-messbar (etwas vereinfacht), falls das Urbild einer abzählbaren Vereinigung von geschlossenen Intervallen wiederum eine abzählbare Vereinigung geschlossener Intervalle ist.

Beispiel 5 Jede differenzierbare Funktion ist Borel-messbar. Denn

$$f^{-1}[x, y] = \bigcup \{[z_1, z_2] : z \in [z_1, z_2] \Leftrightarrow f(z) \in [x, y]\}$$

Zwischen alle diesen Intervalle (die trivial sein können) liegt ein offenes Intervall, also mit echte Ausdehnung. Es ist leicht zu zeigen dass es nur abzählbar viele offene, nicht-überschneidende Intervalle gibt (es gibt nur abzählbar viele ganze Zahlen!)

Nun kommt folgendes Faktum: $\omega_0 \cdot \omega_0 = \omega_0$; sprich: eine abzählbare Vereinigung von abzählbaren Vereinigungen ist eine abzählbare Vereinigung!

Also:

- Jedes Polynom
- Sinus, Kosinus etc.
- Alle Exponentialfunktionen, Fakultät
- Alle Kombinationen hiervon
- ...

Beispiel 6 *Jede Funktion, die differenzierbar ist bis auf endlich viele Punkte ist Borel messbar. Auch bis auf abzählbar viele Punkte? Gute Frage!*

Beispiel 7 *Fraktale, also Funktionen, die stetig, aber in keinem Punkt differenzierbar sind, sind (allgemein) nicht Borel-messbar.*

Beispiel 8 *Die berühmte Wackelfunktion (Dirichlet-Funktion)*

$$(108) f(x) = \begin{cases} 0, & \text{falls } x \in \mathbb{Q} \\ 1 & \text{andernfalls} \end{cases}$$

ist nicht messbar. Denn: $\mathbb{R} - \mathbb{Q} = f^{-1}[0] = \mathbb{R}$ ist natürlich keine abzählbare Vereinigung!

Jetzt sind wir bereit für folgenden Satz:

Satz 9 *Für jede Borel-messbare Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, jedes $\epsilon > 0$, gibt es ein neuronales Netz N mit einem hidden layer, so dass für alle $\vec{z} \in \mathbb{R}^n$ gilt:*

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

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 ist wichtig für Lernbarkeit, nicht Ausdrucksstärke! 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**

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

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.

8.3 Boolesche Funktionen: 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:

$$(109) \quad M_1(1, 0) = (1, 0, -1, 0)$$

$$(110) \quad M_1(0, 1) = (0, 1, 0, -1)$$

$$(111) \quad M_1(1, 1) = (1, 1, -1, -1)$$

$$(112) \quad M_1(0, 0) = (0, 0, 0, 0)$$

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

$$(113) \quad g \circ M_1(1, 0) = (1, 0)$$

$$(114) \quad g \circ M_1(0, 1) = (1, 0)$$

$$(115) \quad g \circ M_1(1, 1) = (1, -1)$$

$$(116) \quad g \circ M_1(0, 0) = (0, 0)$$

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:

$$(117) \quad M_2 \circ g \circ M_1(1, 0) = 1$$

$$(118) \quad M_2 \circ g \circ M_1(0, 1) = 1$$

$$(119) \quad M_2 \circ g \circ M_1(1, 1) = 0$$

$$(120) \quad M_2 \circ g \circ M_1(0, 0) = 0$$

An dieser Stelle ist eine weitere nicht-lineare Funktion unnötig, unser Netz berechnet bereits die XOR-Funktion (auf den passenden Eingaben).

Die Moral von der Geschichte Hier gibt es zwei wichtige Dinge zu merken:

1. Die Breite (des ersten *hidden layers*) liefert uns die nötige Freiheit, um die Eingaben so zu “präparieren”, dass das nicht-lineare Layer damit arbeiten kann. Mit weniger als 4 Neuronen funktioniert das nicht!
2. Die Tiefe des Netzes erlaubt uns, nichtlineare Berechnungen auszuführen. Je tiefer das Netz ist, desto mehr nichtlineare Transformationen werden also hintereinander ausgeführt!

9 Was *lernen* Tiefe Architekturen?

9.1 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 nennt daher den Parameter m auch den **Freiheitsgrad** des Modells. Wir haben also ein Modell

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

Hier sind m_1, m_2 etc. die Freiheitsgrade der Funktion und Hyperparameter, die nicht durch externe Datenformate vorgegeben sind, sondern empirisch optimiert 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*).

Eine wichtige Beobachtung ist folgende: Nimm an, wir haben

- Ein Eingabelayer mit 4 Neuronen
- ein hidden layer mit 5 Neuronen,
- ein Ausgabelayer mit 4 Neuronen

Das macht insgesamt

$$(121) \quad 4 \cdot 5 + 5 \cdot 4 = 40$$

Parameter. Nimm an, wir erweitern das hidden layer auf 10 Neuronen. Dann bekommen wir

$$(122) \quad 4 \cdot 10 + 10 \cdot 4 = 80$$

Parameter. Nimm dagegen an, wir fügen hinzu ein weiteres hidden layer mit 5 Neuronen, also

- Ein Eingabelayer mit 4 Neuronen
- ein hidden layer 1 mit 5 Neuronen,
- ein hidden layer 1 mit 5 Neuronen,
- ein Ausgabelayer mit 4 Neuronen

Dann haben wir

$$(123) \quad 4 \cdot 5 + 5 \cdot 5 + 5 \cdot 4 = 65$$

Parameter. Sprich:

Wenn das Netz in die Tiefe wächst, wächst die Anzahl der Parameter mit den Neuronen langsamer, als wenn das Netz in die Breite wächst!

Es gibt also weniger Freiheitsgrade, also (so grob) bessere Generalisierung.

9.2 (Boolesche) Entscheidungsfunktionen

Klassifikation ist ein erstes Beispiel für induktive Inferenz. Was dabei induziert wird ist eine Funktion f , und zwar eine diskrete Funktion, d.h. eine Funktion die nur endlich viele verschiedene Eingaben nimmt und damit nur endliche viele Ausgaben liefert. Wir werden uns hauptsächlich einen Spezialfall der Klassifikation anschauen, nämlich die **Boolesche Klassifikation**. Boolesche Klassifikation ist deswegen speziell, weil wir eine Boolesche (Wahrheits-)Funktion lernen. Wir suche eine Funktion,

- die für eine Eingabe x entweder “ja” oder “nein” liefert;
- wir fassen “ja” als 1, nein als 0 auf;
- weiterhin basiert eine solche Funktion auf einer Menge von **Attributen**, die auch entweder den Wert 0 oder 1 haben (das werden wir lockern), also erfüllt sind oder nicht.

Wir haben also eine Funktion

$$(124) f : \{0, 1\}^n \rightarrow \{0, 1\}$$

Um mit dem Konzept vertraut zu werden, erstmal folgendes Beispiel (aus Russel & Norvig): es geht um die Entscheidung, ob wir in einem Restaurant warten, bis wir einen Tisch zugewiesen bekommen, oder weitergehen; also eine binäre Entscheidung. NB: wir suchen also unsere eigene Entscheidungsfunktion, möchten also eine Funktion die uns für jedes Restaurant sagt, ob wir warten würden!

Die Attribute sind hier nicht alle binär, aber das tut erstmal nichts zur Sache. Als erstes stellen wir die Liste der Merkmale zusammen, die für unsere Entscheidungsfunktion relevant sind (schöner wäre es natürlich, wenn wir diese Attribute automatisch erstellen könnten, dazu später mehr). Unsere Merkmale sind:

1. Alternativen: gibt es passende Alternativen in der Nähe?
2. Theke: können wir uns an die Theke setzen und schonmal ein Bier trinken?
3. Fr/Sa: ist es Freitag oder Samstag?
4. Betrieb: wie viel Betrieb ist im Lokal? (Werte: leer, einige Leute, voll)
5. Regen: regnet es draußen?
6. Reservierung: haben wir reserviert?
7. Typ: was für eine Art Restaurant haben wir (französisch, italienisch, deutsch)
8. Geschätzte Wartezeit (von uns geschätzt): 0-10,10-30,30-60,>60

Das sind also die Faktoren, die bestimmen, ob wir auf einen freien Tisch warten. Nicht alle Attribute sind binär; wie können sie aber leicht darauf reduzieren; z.B. Attribut 4. kann aufgespalten werden in 2 Attribute: Leer: ja/nein und Voll: ja/nein. Unser Hypothesenraum besteht also aus allen Funktionen

$$(125) \quad h : \{0, 1\}^3 \times \{0, 1, 2\} \times \{0, 1\}^2 \times \{0, 1, 2\} \times \{0, 1, 2, 3\} \rightarrow \{0, 1\}$$

Wie viele solche Funktionen gibt es? Nehmen wir einfachheitshalber mal an, \mathbf{H} wäre die Menge aller Funktionen

$$(126) \quad h' : \{0, 1\}^8 \rightarrow \{0, 1\}$$

Wie groß ist unser Hypothesenraum? Man könnte meinen er wäre nicht übermäßig groß; aber der Eindruck täuscht:

es gibt 2^{2^8} solche Funktionen, also 2^{64}

– eine wahnsinnig große Zahl. Unser Hypothesenraum ist also riesig! Unser Ziel muss es sein, eine möglichst einfache Funktion aus diesem Raum zu wählen, die (nach unseren Begriffen) gut verallgemeinert. Hierbei greift man auf die sogenannten **Entscheidungsbäume** zurück.

Bsp.	Alt	Theke	Fr/Sa	Bet	Reg	Res	Typ	Wart	Warten?
d1	1	0	0	halb	0	1	fr	0-10	1
d2	1	0	0	voll	0	0	it	30-60	0
d3	0	1	0	halb	0	0	de	0-10	1
d4	1	0	1	voll	1	0	it	10-30	1
d5	1	0	1	voll	0	1	fr	>60	0
...									

Table 1: Ein Ausschnitt aus unserem Datensatz

9.3 Entscheidungsbäume

Boolesche Funktionen lassen sich einfach als Tabellen auffassen; wir nehmen nun wieder unser Beispiel, um das darzustellen: Tabelle 1 ist nur ein kleiner Ausschnitt unserer Funktion; wir können auch annehmen, es handelt sich um unseren Datensatz D . Ein **Entscheidungsbaum** ist einfach ein Baum,

1. in dem jeder Knoten ein Merkmal repräsentiert,
2. jedes Blatt einen Wert, den die Funktion annimmt;
3. auf jedem Pfad von der Wurzel zu einem Blatt kommt dabei jedes Merkmal höchstens einmal vor.

Jede Boolesche Funktion lässt sich als Entscheidungsbaum darstellen: wir können einfach den Baum nehmen, in dem jede *Zeile* unserer Tabelle einem *Pfad* entspricht. Es gibt gewisse Boolesche Funktionen, die lassen sich nicht oder nur sehr schwer kompakt repräsentieren, z.B.

die **Paritätsfunktion** (f nimmt den Wert 1 an, wenn eine gerade Zahl von Argumenten den Wert 1 annimmt), oder

die **Majoritätsfunktion** (f nimmt den Wert 1 an, falls mindestens die Hälfte seiner Argumente den Wert 1 annimmt).

Allerdings gibt es auch Entscheidungsbäume, die eine wesentlich kompaktere Darstellung erlauben. Wenn wir das obige Beispiel betrachten, dann fällt uns z.B. auf dass wann immer die geschätzte Wartezeit >60 Minuten beträgt, dann warten wir niemals darauf dass ein Tisch frei wird. Wenn sich dieses Muster durch alle unsere Beobachtungen zieht, dann können wir also

dieses Merkmal an die Wurzel unseres Baumes setzen, und dann können wir in einigen Fällen den Baum an dieser Stelle schon mit dem Blatt 0 beenden. Algorithmen zur Induktion von Entscheidungsbäumen beruhen genau auf dieser Beobachtung:

Wir können die Komplexität von Booleschen Funktionen messen nach der Komplexität der Entscheidungsbäume.

Das wiederum passt zu unseren obigen Beobachtung, dass einfache Funktionen eher sinnvolle, interessante Generalisierungen liefern als komplexe. Wir bekommen also folgendes:

Gegeben eine Menge D von Daten, finde den einfachsten Entscheidungsbaum, der mit D konsistent ist; die zugehörige Boolesche Funktion ist unsere Hypothese h .

9.4 Neuronale Netze als Entscheidungsbäume

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 layers haben, dann können wir uns das vorstellen als einen (stark erweiterten) Entscheidungsbaum mit n Ebenen. Erweitert, 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

$$(127) \quad B_{M_i=0}$$

$$(128) \quad B_{M_i=1}$$

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

$$(129) \quad \vec{x} = (x_1, \dots, x_{i-1}, a, x_{i+1}, \dots, x_n)$$

und das lineare Modell mit

$$(130) \quad M = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,i} & \dots & x_{1,n} \\ \dots & & & & & \end{pmatrix}$$

wobei wir setzen:

$$(131) \quad x_{1,1} = x_{1,2} = \dots = x_{1,i-1} = x_{1,i+1} = \dots = x_{1,n} = 0$$

$$(132) \quad x_{1,i} = 1$$

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

$$(133) \quad \vec{z}_j = \max(y_1, y_k)$$

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.

10 Optimierung von Neuronalen Netzen

10.1 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.

10.2 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:

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

Allgemeiner:

Definition 10 Eine Menge $X \subseteq \mathbb{R}^n$ ist konvex, falls gilt: für alle Skalare $\lambda \in [0, 1]$, $\vec{x}, \vec{y} \in X$,

$$\lambda \vec{x} + (1 - \lambda) \vec{y} \in X.$$

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

$$(134) \quad \overline{xy} = \{\lambda \vec{x} + (1 - \lambda) \vec{y} : \lambda \in [0, 1]\}$$

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**

$$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

ist konvex, falls

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

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

$$(135) \quad \operatorname{argmin} f(x)$$

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

$$(136) \quad f(x) > f(x + \epsilon) < f(x + 2\epsilon)$$

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

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

$$(137) \quad \operatorname{argmin}_{x \in X} f(x)$$

zu finden. Das Optimierungsproblem 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

$$f, g_1, \dots, g_n : \mathbb{R}^n \rightarrow \mathbb{R},$$

und wir suchen

$$(138) \quad \operatorname{argmin}_{x \in \mathbb{R}^n} f(x), \text{ vorausgesetzt dass } g_1(x), g_2(x), \dots, g_i(x) \leq 0$$

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

11 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:

$$(139) \quad \frac{df}{dx_i} f(x_1, \dots, x_i, \dots, x_n) = \frac{df}{dx} f(a_1, \dots, x, \dots, a_n)$$

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

$$(140) \quad \frac{df}{dx} x^2 + 2y = 2x$$

und

$$(141) \quad \frac{df}{dy} x^2 + 2y = 2$$

Allerdings gilt:

$$(142) \quad \frac{df}{dx} x^2 \cdot 2y = 2x \cdot 2y$$

und

$$(143) \quad \frac{df}{dy} x^2 \cdot 2y = 2x^2$$

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.

12 Gradientenbasierte Optimierung

12.1 Das Problem in höheren Dimensionen

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

$$(144) \operatorname{argmin}_{\vec{x} \in \mathbb{R}^n} f(\vec{x})$$

suchen. Der Grund ist folgender: wenn wir

$$(145) \operatorname{argmin}_{x \in \mathbb{R}} f(x)$$

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

$$(146) f(a) > f(a + l)$$

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

$$(147) f(a + l) > f(a + l + l)$$

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

$$(148) f(\vec{a}) > f(\vec{a} + \vec{b}l)$$

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

$$(149) \quad f(\vec{a}) > f(\vec{a} - \nabla f(\vec{a})l)$$

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

- Der Gradient gibt uns den steilsten Anstieg für eine infinitesimale Veränderung, d.h. es ist nicht gesagt dass $f(\vec{a}) < f(\vec{a} + \nabla f(\vec{a})l)$. 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, \dots mit $\lim_{n \rightarrow \infty} l_n = 0$, dann werden wir ein lokales Minimum finden!

12.2 Die effektive Prozedur

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

$$(150) \operatorname{argmin}_{x_1, \dots, x_n \in \mathbb{R}^n} f(x_1, \dots, x_n)$$

gegeben dass

$$(151) \quad g_1(x_1, \dots, x_n) < 0$$

$$(152) \quad \dots$$

$$(153) \quad g_i(x_1, \dots, x_n) < 0$$

Hier sind g_1 bis g_n die Parameter, wahren 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 $(0, \dots, 0)$ 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 (relativ) beliebigen Punkt

$$\vec{a} = (a_1, \dots, a_n) \in \mathbb{R}^n$$

an; der Punkt wird zunächst mehr oder weniger zufällig ausgewählt. Als nächstes nehmen wir ∇f , und setzen:

$$(154) \vec{a}' := \vec{a} - l\nabla f(\vec{a})$$

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 154 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:

$$(155) \vec{a}'' := \vec{a}' - l\nabla f(\vec{a}')$$

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 Minimum ist.

Was ebenfalls ein großer Vorteil ist, ist falls

$$(156) \quad \nabla^n f = 0$$

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

13 Differenzieren – einige Faustregeln

Differenzierung ist vornehm für Ableitung; hier wollen wir einige Ableitungsregeln besprechen, damit wir verstehen was hinterher passiert.

Die Additionsregel Zunächst folgende **Additionsregel**:

$$(157) (f(x) + (g(x)))' = f(x)' + g(x)'$$

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

$$(158) \frac{\partial f}{\partial x}(f(x, \vec{y}) + g(x, \vec{y})) = \frac{\partial f}{\partial x}f(x, \vec{y}) + \frac{\partial f}{\partial x}g(x, \vec{y})$$

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

$$(159) \nabla(f(\vec{x}) + g(\vec{x})) = \nabla f(\vec{x}) + \nabla g(\vec{x})$$

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

$$(160) f \circ g(x) = f(g(x))$$

Die Regel lautet:

$$(161) (f \circ g(x))' = f'(g(x)) \cdot g'(x)$$

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

$$(162) \left(\frac{1}{x}\right)' = -\frac{1}{x^2}$$

Wenn wir nun die Ableitung von

$$(163) f(x) = \frac{1}{3x^2}$$

berechnen wollen, geht das mittels

$$(164) \quad f(x) = \frac{1}{x} \circ 3x^2(x)$$

Also haben wir

$$(165) \quad f'(x) = -\frac{1}{(3x^2)^2} \cdot 6x = -\frac{6x}{9x^4} = -\frac{2}{3x^3}$$

Wo kommt diese Regel her? Erinnern wir uns das

$$(166) \quad f'(x) = \lim_{x \rightarrow 0} \frac{f(x)}{x}$$

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

$$(167) \quad \lim_{x \rightarrow 0} \frac{g \circ f(x)}{x} = \lim_{x \rightarrow 0} \frac{g \circ f(x)}{f(x)} \frac{f(x)}{x}$$

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

$$(168) \quad \lim_{x \rightarrow 0} \frac{g(x)}{f(x)} = g'(f(x))$$

$$(169) \quad \lim_{x \rightarrow 0} \frac{f(x)}{x} = f'(x)$$

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.

Kettenregel für partielle Ableitung Die Kettenregel gilt auch bei multivariaten Funktion (aus demselben Grund wie bei einfachen Funktionen) und für partielle Ableitungen. Sei x eine Variable einer multivariaten Funktion, \bar{y} die übrigen Variablen. Es gilt:

$$(170) \quad \frac{d(f \circ g)}{dx} f \circ g(x, \bar{y}) = \left(\frac{df}{dx} f\right) \circ g(x, \bar{y}) \cdot \frac{dg}{dx} g(x, \bar{y})$$

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

$$(171) \quad \frac{df}{dx} \frac{1}{1 + e^{-x}} = \frac{e^x}{(e^x + 1)^2} = \frac{1}{1 + e^{-x}} \cdot \left(1 - \frac{1}{1 + e^{-x}}\right)$$

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

$$(172) \quad S'(x) = S(x) \cdot (1 - S(x))$$

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

14 Tiefe Netze trainieren

14.1 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 (ReLU, maxout, Sigmoid,...)
- Die Größe der linearen layer (= Anzahl der Zeilen der Matrizen)
- Die Natur der Ausgabefunktion (softmax, max,...)

Parameter sind:

- Die Werte der Matrizen in den linearen Layern.
- Je nach nichtlinearen layern: z.B. die Gruppen der maxout-unit, die Parameter für parametrisiertes ReLU (betrachten wir hier nicht).

Wie üblich denotieren wir die **Gesamtheit der Parameter** mit

$$\theta$$

Wenn wir also für ein Modell die Hyperparameter festlegen und das Netz N nennen, dann ist

$$N_\theta$$

eine Instanziierung von der Parameter mit θ . Wir schreiben das aber meistens nicht aus.

Wenn wir also tiefe Netze trainieren, dann bekommen wir ein **Optimierungsproblem**. Wir haben dabei gegeben:

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

Wir messen also

$$(173) \sum_{j=1}^i K(N_{\theta}(\vec{x}_j), y_j)$$

als unsere Gesamtkosten; das ist der Term den wir minimieren möchten.

Minimieren heißt: über die Parameter θ von N . Wir müssen das Modell also ausbuchstabieren:

$$(174) N_{\theta}(\vec{x}) = g_{out} \circ M_k \circ g_{k-1} \circ M_{k-1} \circ \dots \circ g_1 \circ M_1(\vec{x})$$

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

$$(175) \underset{M_k, M_{k-1}, \dots, M_1}{\operatorname{argmin}} \sum_{j=1}^i K(g_{out} \circ M_k \circ g_{k-1} \circ M_{k-1} \circ \dots \circ g_1 \circ M_1(\vec{x}_j), y_j)$$

Es ist natürlich klar, dass man für 175 keine analytische Lösung findet, es ist bereits schwer genug, einen Gradienten zu finden. Dafür fasst 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:

⇒ die Funktion in vielen Variablen hat einen Gradienten. Den Versuchen wir zu verringern.

Übersicht Das Training zerfällt in drei Teile:

- Batching
- Backpropagation
- Optimierung (Parameter update)

Die Ausgabe jedes Schrittes ist die Eingabe des nächsten; jede Trainingsepoche durchläuft alle drei.

Batching ist eine Art sampling; aus dem Datensatz wird eine Teilmenge ausgewählt, auf der wir optimieren.

Backpropagation ist ein Algorithmus, um den Gradienten zu bestimmen. Wir können damit den Gradienten für beliebige Parameter (und die Eingabe) der Funktion bestimmen.

Der eigentliche Optimierungsschritt besteht darin, die Parameter neu zu setzen. Der einfachste Algorithmus ist stochastic gradient descent, in praktischen Anwendungen sind aber mittlerweile andere Algorithmen, so wie Adam und Adamax ebenso geläufig.

Batching Ein zentraler Punkt ist: man optimiert neuronale Netze nicht über den gesamten Datensatz (zu aufwändig), und auch nicht über einem einzelnen Datenpunkt (zu fehleranfällig), sondern über einer normalerweise recht kleinen Teilmenge, die **batch**. Optimierung läuft nicht sequentiell über die Datenpunkte der batch, sondern ein Optimierungsschritt berücksichtigt die gesamte Batch. Die Größe der batch ist ein Hyperparameter. Also:

$$(176) \quad B \subseteq D, |B| = n < i$$

Der Punkt ist hier: wir führen die Optimierung *nicht sequentiell* für die Batch aus, sondern *parallel*:

- ▶ Anstatt dem Netz eine Eingabe \vec{x} zu geben, geben wir eine Matrix $M = (\vec{x}_1, \dots, \vec{x}_i)$
- ▶ Wir definieren nun die parallele Berechnung: $N((\vec{x}_1, \dots, \vec{x}_i)) = (N(\vec{x}_1), \dots, N(\vec{x}_i))$
– die Berechnungen des Netzes werden parallel ausgeführt.
- ▶ Wenn wir annehmen, die Ausgabe des Netzes war vorher eine Zahl y , bekommen wir nun einen Vektor (y_1, \dots, y_i)
- ▶ Gleichzeitig hatten wir vorher unsere Zielausgaben \hat{y} ; die werden nun zusammengefasst zu einem Vektor $(\hat{y}_1, \dots, \hat{y}_i)$
- ▶ Wir berechnen dann die Kosten als die (euklidische) Distanz der Vektoren $d((\hat{y}_1, \dots, \hat{y}_i), (y_1, \dots, y_i))$

Der Term, den wir in einer Trainingsepoche optimieren wollen ist also θ in:

$$(177) \quad d(N_\theta(M), (\hat{y}_1, \dots, \hat{y}_i))$$

14.2 Backpropagation

Backpropagation ist 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!) Backpropagation ist also noch kein Optimierungsalgorithmus! Backpropagation liefert uns den Gradienten über die *Eingabe* des Netzes, aber auch über beliebige Parameter.

⇒ Wir kriegen also für jeden Punkt/Parameter des Netzes die partielle Ableitung.

Backpropagation nimmt im Rahmen von Deep Learning als Eingabe immer die ganze Batch. Dennoch stellen wir hier den Algorithmus vor für eine einzige Eingabe vor. Wir haben also

- Eingabe \vec{x}
- Neuronales Netz N_θ (alles bereits initialisiert)
- Zielausgabe \hat{y} (wir nehmen tatsächlich an es ist eine Zahl, kein Vektor!)
- Kostenfunktion K

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

$$(178) \quad K(N_\theta(\vec{x}), \hat{y}) = F_{\hat{y}}(\vec{x})$$

Wir versuchen den Gradienten von F auf einem bestimmten Datenpunkt zu berechnen, um uns somit der Nullstelle der Funktion zu nähern. Da

$$(179) \quad F_{\hat{y}} : \mathbb{R}^n \rightarrow \mathbb{R}_0^+$$

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

Man beachte auch: wir haben keine Vertauschung von Argument und Parametern vorgenommen, wie etwa bei linearer Regression!

14.3 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, für Trainingsdaten $\{(\vec{x}_i, \vec{y}_i) : i \in I\}$

$$(180) \quad F = \frac{1}{2} \sum_{i=1}^n \|N(\vec{x}_i) - \vec{y}_i\|_2$$

Wichtig ist hierbei, dass $F : \mathbb{R}^n \rightarrow \mathbb{R}_0^+$, 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 erstmal an, dass die Aktivierungsfunktionen g feststehen, also nicht optimiert werden. Es geht also nur um die linearen Modelle.

\Rightarrow 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 $o \in \mathbb{R}$!

14.4 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 \hat{y} , die wir benötigen um die Kostenfunktion zu instantiieren; so haben wir F . Wir nehmen an, \hat{y} ist eine Zahl. Das ist nur Einfachheit halber; praktisch ist das oft nicht gegeben, macht aber auch keine weiteren Schwierigkeiten.

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

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

Wir werden jetzt den Algorithmus erklären, allerdings nicht mittels Pseudocode.

14.5 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 nichtlinearen 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!

14.6 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
 1. eine Menge von Knoten, die eine Eingabe liefern, und
 2. 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:

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

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!

14.7 Berechnungsgraphen – Knoten und Funktionen

Erinnern wir uns, dass im Berechnungsgraphen ein Knoten v einer (atomaren) Funktion f_v entspricht. Wir nehmen nun für jeden Knoten v **zwei** Funktionen $r(v), l(v)$

1. $r(v) = f_v$

2. $l(v) = f'_v$

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

14.8 Backpropagation – Der Feedforward-Step

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

Fall 0 Die Eingabeknoten

Der Eingabeknoten v_i bekommt

$$(181) \quad r(v_i) = x_i$$

$$(182) \quad l(v_i) = 1$$

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 $|\uparrow v| = 1$

$$(183) \quad r(v) = f_v(r(\uparrow v))$$

$$(184) \quad l(v) = f'_v(r(\uparrow v))$$

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

$$f(x) = wx,$$

dann ist

$$f'_v(x) = w,$$

also $l(v) = w$ – denn es kommt keine Variable darin vor!

Falls v der Sigmoidfunktion entspricht, dann wird links die Ableitung eingetragen:

$$S'(x) = S(x) \cdot (1 - S(x)).$$

Fall 2 : $f_v = x_1 + \dots + x_i$

Der Knoten hat also eine Menge von Vorgängern.

$$(185) \quad r(v) = \sum_{v' \in \uparrow v} r(v')$$

$$(186) \quad l(v) = (1, \dots, 1)$$

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

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

$$(187) \quad r(v) = F(\vec{x})$$

$$(188) \quad l(v) = (1, \dots, 1)$$

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.

14.9 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

$$(189) \quad \vec{z} = \nabla F(\vec{x})$$

Am Ende des Netzes F werden die Kosten berechnet (meistens eine Summe (bei Vektoren) aber nicht notwendig (wenn die Ausgabe eine Zahl ist!)).

Fall 0 Der Ausgabeknoten

Der Ausgabeknoten ist gesetzt mit

$$(190) \quad l(v_o)$$

Nun gehen wir von hinten nach vorne durch. Die Ausgabe der Netzes wird also von hinten wieder nach vorne durchgereicht und modifiziert!

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:

$$(191) \quad l(v) := l(v) \cdot l(\downarrow v)$$

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:

$$(192) \quad l(v) := l(v) \sum_{v' \in \downarrow v} l(v')$$

Fall 3 $|\uparrow v| > 1$

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

$$(193) \quad l(v) := (l(\downarrow v), \dots, l(\downarrow v))$$

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 .

14.10 Backpropagation: Ausgabe

Nachdem wir den FF und den BP-Schritt gemacht haben, haben wir für jeden Knoten v_i in unserem Berechnungsgraphen zwei Zahlen:

$$v \mapsto (l(v_i), r(v_i))$$

Wir erinnern uns dass $F(\vec{x}, \hat{y}) = K(N(\vec{x}), \hat{y})$. Was wir nun zeigen müssen ist:

Satz 11 *Sei v_θ ein Knoten im Berechnungsgraphen, der einem Parameter θ des Netzes N entspricht. Dann gilt:*

$$l(v_\theta) = \frac{d\theta}{dF} F(\vec{x}, \hat{y})$$

Wir können also – wenn das stimmt – effektiv den Gradienten ausrechnen für jeden Parameter – hurra!

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

14.11 Beweis von Satz 11

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, \dots, v_n ,
2. Ausgabeknoten o_1, \dots, o_k .

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

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

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:

$$(194) \quad \nabla(w \cdot f(x)) = w \nabla f(x)$$

(Produktregel für Ableitungen/Gradienten!)

Als nächstes werden die Werte summiert; hier haben wir

$$(195) \quad \nabla(f_1(\vec{x}) + \dots + f_k(\vec{x})) = \nabla f_1(\vec{x}) + \dots + \nabla f_k(\vec{x})$$

(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 nichtlineare Funktion g mit Ableitung g' . Wir notieren in dem Knoten also den Wert

$$(196) \quad g'(f(\vec{x})),$$

$f(\vec{x})$ 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

$$(197) \quad f_{n1}(\vec{x}) = g(w_1 f_{o1}(\vec{x}) + \dots + w_k f_{ok}(\vec{x}))$$

Dementsprechend ist nach Kettenregel

$$(198) \quad f'_{n1}(\vec{x}) = g'(w_1 f_{o1}(\vec{x}) + \dots + w_k f_{ok}(\vec{x})) \cdot (w_1 f'_{o1}(\vec{x}) + \dots + w_k f'_{ok}(\vec{x}))$$

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

1. Er nimmt die Eingabe 1, multipliziert sie mit $g'(w_1 f_{o1}(\vec{x}) + \dots + w_k f_{ok}(\vec{x}))$.
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

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

was genau einer Anwendung der Kettenregel entspricht.

Das funktioniert 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!

14.12 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.

Wir erinnern uns aber: wir haben unser Netz N_θ mit den Parametern

$$(200) \quad \theta = M_1, \dots, M_i$$

Es sind ja hier die linearen Layer die trainiert werden. Wir denotieren diese Parameter mit θ in ihrer Gesamtheit. Mittels backpropagation berechnen wir

$$(201) \quad \nabla_\theta K(\vec{x}, \hat{y})$$

Gradient descent macht nun einfach folgendes: wir setzen

$$(202) \quad \theta' = \theta - \ell \left(\nabla_\theta \frac{1}{m} K(\vec{x}, \hat{y}) \right)$$

wobei ℓ die Lernrate ist. Also: wir subtrahieren den Gradienten, multipliziert mit der Lernrate, von den Parametern.

14.13 Alternative: Momentum

Moderne Algorithmen (wie Adam, Adamax) basieren auf Momentum, einem Begriff der aus der Physik stammt (Masse \cdot Geschwindigkeit).

Lernen mit Momentum soll die Abhängigkeit von womöglich stark wechselnden Gradienten verringern (Schluchtbeispiel).

Stattdessen soll aus wechselnden Richtungen eine generelle Richtung extrapoliert werden:

$$(203) \quad \vec{v}' := \alpha \vec{v} - l \nabla_{\theta} \frac{1}{m} K(\vec{x}, \hat{y})$$

$$(204) \quad \theta' = \theta + \vec{v}'$$

Hier ist \vec{v} das Momentum (also die alte Bewegungsrichtung), $\alpha \in [0, 1)$ ein Hyperparameter, der angibt, wie stark das alte Momentum einfließt in das neue (also sich verringert in jedem Schritt). Das Momentum wird errechnet aus dem Gradienten, und erst das Momentum fließt direkt ein in die neuen Parameter.

Adam bedeutet: adaptive momentum. Adam und Adamax basieren also auf diesem Begriff!

14.14 Zusammenfassung

Man kann das Training eines neuronalen Netzes also in drei Teile aufspalten:

1. Batching – aus dem Datensatz wird eine (üblicherweise kleine) Teilmenge genommen, die Batch, mit der eine Trainingsepoche durchgeführt wird.
2. Mit der Batch und dem Netz wird nun Backpropagation durchgeführt. Die Ausgabe dieses Schrittes ist der Gradient des Netzes für die Batch!
3. Mit dem errechneten Gradienten werden die Parameter neu gesetzt: $\theta := \theta - l\nabla_{\theta}F$. Wir speichern diese neuen Parameter.

Das ist eine Trainingsepoche.

Wichtig ist aber: SGD ist im Deep Learning keinesfalls dominant; es viele weitere gradientenbasierte Optimierungsalgorithmen, die ebenfalls sehr geläufig sind.

15 Initialisierung

DL-Algorithmen basieren auf Iterationen, haben keine Garantie der Konvergenz, und selbst wenn sie konvergieren gibt es keine Garantie dass wir ein globales Minimum erreicht haben. Es ist leicht zu sehen dass unter solchen Bedingungen die Initialisierung des Netzes – also bildhaft gesprochen: der Punkt von wo wir die Reise durch den Parameterraum starten – eine große Rolle spielt.

Zentral: keine gleich initialisierten Parameter, sonst werden die Parameter auch gleich auf alle Eingaben reagieren. Daher am besten eine zufällig gewählte Initialisierung (Gleichheit extrem unwahrscheinlich).

Hier zwei Möglichkeiten:

1. Gezogen von der uniformen Verteilung (über ein begrenztes Intervall, normalerweise $[0, 1]$)
2. Gezogen von einer Normalverteilung (mit Erwartungswert 0)

Was ist besser? Hier folgendes Bedenken:

1. Uniforme Verteilung setzt alle Zahlen gleich, also auch Randwerte treten häufig auf. Andererseits haben wir eine *a priori* Beschränkung des Intervalls.
2. Normalverteilung heißt: wir tendieren eher zu kleinen Zahlen (bei $\mu = 0$, wie üblich), haben aber theoretisch keine Beschränkung – praktisch aber schon.

Es hilft sich auch klarzumachen, was große/kleine initialisierte Werte bedeuten in Begriffen der Wahrscheinlichkeitstheorie:

- Kleine Initialwerte bedeuten dass wir apriori annehmen, dass das entsprechende Merkmal (d.h. entsprechende Eingabekomponente) fast keinen Einfluss auf die Klassifizierung hat. Um also einen deutlichen Einfluss zu nehmen brauchen wir Evidenz, die bekommen wir über große Gradienten.
- Große Initialwerte bedeuten dass wir apriori einen großen Einfluss auf die Klassifizierung annehmen, und Evidenz muss erst zeigen, dass es eigentlich irrelevant ist.

Insgesamt ist aber die Interaktion dieser Komponenten nicht wirklich gut verstanden, soweit ich weiss.

16 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 Modell)

$$(205) \quad L(\vec{x}) = M\vec{x} + \vec{a}$$

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).

Ein spezielles Verfahren der Regularisierung mit neuronalen Netzen ist der sog. **Dropout**.

- Im Dropout werden einfach mal so ein bestimmter Anteil der Verbindungen von Neuronen gekappt.
- Das bedeutet dass, wenn wir das ganze als Matrixmultiplikation darstellen, der entsprechende Eintrag in der Matrix auf 0 gesetzt wird;

Unsere Matrix wird also **dünn**.

Diese Form der Regularisierung ist wohl schwierig theoretisch zu rechtfertigen, aber funktioniert in der Praxis recht gut. Das ist wohl am ehesten so zu erklären: durch den Wegfall von Einträgen der einzelnen Matrizen wird der Fit der einzelnen Layer schlechter, die Anzahl der Parameter drastisch reduziert, während die Tiefe und Anzahl der Neuronen beibehalten wird. Es scheint dass diese letzten beiden Faktoren entscheidend beitragen zur Generalisierung, die damit tendenziell verbessert wird.

17 Word embeddings

17.1 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 je nachdem eine Funktion

$$(206) \quad \text{vec} : \Sigma^* \rightarrow \mathbb{R}^n$$

$$(207) \quad \text{vec} : \Sigma \rightarrow \mathbb{R}^n$$

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.

17.2 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*. Die Ordnung interessiert uns dabei nicht, sondern nur die Frage: kommen w, w' im selben Skip-gram vor?

Formaler, wir möchten den folgenden Term maximieren:

$$(208) \quad \frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c \& j \neq i} \log P(w_{t+j} | w_t)$$

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

$$(209) \quad w_1 w_2 \dots w_{10}$$

dann wird unser Datensatz:

$$(w_1, w_2), (w_1, w_3), (w_2, w_1), (w_2, w_3), (w_2, w_4), (w_3, w_1) \dots$$

wobei wir natürlich die Zahl der Vorkommen uns merken müssen. Was wir nun suchen ist: eine Vektorrepräsentation 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

$$Lex = \{w_1, \dots, w_n\}$$

Dann nehmen wir

$$(210) \quad vec_1(w_i) = (\overbrace{0, \dots, 0}^{i-1 \text{ mal}}, 1, \overbrace{0, \dots, 0}^{n-i \text{ mal}})$$

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

$$(211) \quad \vec{v} = vec_1(\operatorname{argmax}_v P_{skip-gram}(w|v))$$

wobei $P_{skip-gram}$ einfach die Wahrscheinlichkeit liefert, dass w im Skip-gram 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

$$(212) \quad g \circ M$$

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 :

$$(213) \quad Mvec_1(w) = M_{-,i}$$

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). Aber Vorsicht:

$$Mvec_1 w \in \mathbb{R}^{300} \text{ (mehr oder weniger),}$$

das bedeutet: $softmax(Mvec_1 w \in \mathbb{R}^{300})$ kann diese Wahrscheinlichkeiten *nicht* liefern – wir brauchen ja eine Verteilung über $|Lex|^2$ Ereignisse – und das kann leicht in der Größenordnung 10^{10} sein!

Man muss also etwas komplexeres finden. Hier repräsentiert M das lineare layer.

$$(214) \quad P_N(w|v) = \frac{\exp(Mvec_1(w)^\top Mvec_1(v))}{\sum_{w' \in Lex} \exp(Mvec_1(w')^\top Mvec_1(v))}$$

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

$$(215) \quad \nabla \log(P(w|v))$$

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 214 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:

$$(216) \quad \hat{P}(w|v) = \frac{\text{Anzahl der Vorkommen } (w, v) \text{ in den Daten}}{\text{Anzahl der Vorkommen } (x, v) \text{ in den Daten (beliebiges } x)}$$

Wir können daraus die Kostenfunktion ableiten:

$$(217) \quad K(w, v) = \frac{1}{2} N(w, v) - \hat{P}(w|v)$$

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:

$$(218) \quad \text{vec}(w) = M' \text{vec}_1(w)$$

Anders gesagt: jedes Wort w wird also eingebettet als die *Spalte* der Matrix des Modells, die von $\text{vec}_0(w)$ 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, Kookkurrenzen 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.

17.3 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

$$(219) F : \Sigma^* \times \Sigma^* \rightarrow [0, 1]$$

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:

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

Also gilt: $n(w, 1)$ ist die Wurzel, $n(w, L(w)) = w$. Sei x ein beliebiger innerer Knoten des Baumes. Wir lassen $\text{succ}(x)$ einen beliebigen, 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

$$(220) \quad \|t = s\| = \begin{cases} 1, & \text{falls } t = s \text{ wahr ist} \\ -1 & \text{andernfalls .} \end{cases}$$

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:

$$(221) \quad P(w|v) = \prod_{j=1}^{L(w)-1} S(\|n(w, j+1) = \text{succ}(n(w, j))\| \cdot \text{Mvec}(n(w, j))^{\top} \text{Mvec}(v))$$

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?

17.4 Sog. “Additive Kompositionalität”

Ein berühmtes praktisches Beispiel ist folgendes:

$$(222) \quad (\text{vec}(\text{Madrid}) - \text{vec}(\text{Spain})) + \text{vec}(\text{France}) \approx \text{vec}(\text{Paris})$$

wobei \approx bedeutet: es gibt kein $w \in \text{Lex}$ so dass $\text{vec}(w)$ näher an $\text{vec}(\text{Paris})$ liegt als der Vektor in (222).

Eine wichtige Eigenschaft von Einbettungen ist also folgende: wir möchten semantische Relationen in numerische Übersetzen können semantische Verwandtschaft 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.:

$$(223) \quad \text{vec}(\text{kleines Auto}) \approx \text{vec}(\text{kleines}) + \text{vec}(\text{Auto})$$

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

$$(224) \quad \vec{x}^\top \vec{y} = |\vec{x}| \cdot |\vec{y}| \cdot \cos(\angle(\vec{x}, \vec{y}))$$

wobei $\angle(\vec{x}, \vec{y})$ den Winkel der beiden Vektoren bezeichnet. Also:

$$(225) \quad \cos(\angle(\vec{x}, \vec{y})) = \frac{\vec{x}^\top \vec{y}}{|\vec{x}| \cdot |\vec{y}|}$$

Wir haben (erinnern wir uns)

$$(226) \quad \cos(\angle) = \frac{\text{Ankathete von } \angle}{\text{Hypotenuse}}$$

im rechtwinkligen Dreieck, also:

$$(227) \quad \cos(0^\circ) = 1$$

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 (Terminformung). 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.

18 Recurrent Neural Networks

18.1 Motivation

Ein großer Nachteil von neuronalen Netzen ist, dass die Eingaben eine festgelegte Länge, d.h. Anzahl von Komponenten haben. Oftmals wollen wir aber Sequenzen verarbeiten, die beliebig lang sind! Ein weiteres Problem ist, dass 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 unbekanntem 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.

18.2 RNN – zwei Auffassungen

Mathematisch gesehen ist ein RNN eine Funktion

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

wobei 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

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

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

$$RNN \text{ bzw. } RNN^+$$

Wohlgemerkt: das Netz kann in beiden Fällen vollkommen gleich aussehen, wir interpretieren es nur anders.

18.3 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_t .

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_{i+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:

$$(228) \quad RNN^+(\vec{x}_1, \dots, \vec{x}_n, s_0) = (\vec{y}_1, \dots, \vec{y}_n)$$

$$(229) \quad RNN(\vec{x}_1, \dots, \vec{x}_n, s_0) = \vec{y}_n$$

wobei

$$(230) \quad \vec{y}_i = O(\vec{s}_i)$$

$$(231) \quad \vec{s}_i = N(\vec{s}_{i-1}, \vec{x}_i)$$

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 ebenfalls
2. Wie kommen die Vektoren \vec{x}, \vec{s} zusammen in N ? Antwort: verschiedene Möglichkeiten, z.B. Addition, oder aber zwei Eingabelayer, also im Prinzip dasselbe wie eine Eingabe $\vec{x}' = (\vec{x}, \vec{s})$.
3. Was sind die Parameter? Das sind immer dieselben, durch alle Berechnungsschritte.

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

$$(232) \quad s_4 = N(s_3, x_4)$$

$$(233) \quad = N(N(s_2, x_3), x_4)$$

$$(234) \quad = N(N(N(s_1, x_2), x_3), x_4)$$

$$(235) \quad = N(N(N(N(s_0, x_1), x_2), x_3), x_4)$$

Ebenso gilt:

$$\begin{aligned} (236) \quad y_4 &= O(N(s_3, x_4)) \\ (237) \quad &= O(N(N(s_2, x_3), x_4)) \\ (238) \quad &= O(N(N(N(s_1, x_2), x_3), x_4)) \\ (239) \quad &= O(N(N(N(N(s_0, x_1), x_2), x_3), x_4)) \end{aligned}$$

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!

18.4 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:

$$(240) \quad s_i = s_{i-1} + x_i$$

$$(241) \quad y_i = s_i$$

wobei

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

Wenn wir nun s_0 – den “Startzustand” – entsprechend setzen als

$$(242) \quad s_0 = (0, \dots, 0)$$

Dann bekommen wir, für eine Eingabe

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

die Ausgabe

$$(243) \quad \vec{y}_m = \sum_{j=1}^m \vec{x}_j$$

Wie summieren also, im Falle einer Eingabe

$$w_1 \dots w_n$$

einfach die Vektoren

$$\text{vec}(w_1) + \dots + \text{vec}(w_n)$$

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

18.5 SRNN

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

$$(244) \quad s_i = N_{SRNN}(x_i, s_{i-1}) = g(Ms_{i-1} + M'x_i + \vec{a})$$

$$(245) \quad y_i = O_{SRNN}(s_i) = s_i$$

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

$$(246) \quad g = \frac{1}{1 + e^{-x}}$$

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.

18.6 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, \dots, N_i die Funktionen des ersten, ..., *i*ten Layers
- O die Ausgabefunktion

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

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

Dann ist also:

$$(247) \quad \vec{y}_1 = O(N_i(\dots(N_1(\vec{x}_1, \vec{s}_0^1)\dots), \vec{s}_0^i))$$

Desweiteren bekommen wir eine Liste von Zuständen

$$(248) \quad \vec{s}_1^1 = N_1(\vec{x}_1, \vec{s}_0^1)$$

$$(249) \quad \vec{s}_1^2 = N_2(N_1(\vec{x}_1, \vec{s}_0^1), \vec{s}_0^2)$$

$$(250) \quad \vdots$$

$$(251) \quad \vec{s}_1^i = N_i(\dots(N_2(N_1(\vec{x}_1, \vec{s}_0^1), \vec{s}_0^2)\dots), \vec{s}_0^i)$$

Wir berechnen im nächsten Schritt also

$$(252) \quad \vec{y}_2 = O(N_i(\dots(N_1(\vec{x}_2, \vec{s}_1^1)\dots), \vec{s}_1^i))$$

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.

18.7 Training von RNNs

RNNs einfallen/ausfallen Wichtig ist: nehmen wir ein RNN, das eine Eingabe

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

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 $(\vec{x}_1, \dots, \vec{x}_n)$ 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 bisheriger Backpropagation-Algorithmus nicht!

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

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

$$(253) \quad F(N(\vec{x}_1, \dots, \vec{x}_{n-1}), \vec{x}_n)$$

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

$$(254) \quad \frac{\partial F(N(\vec{x}_1, \dots, \vec{x}_{n-1}), \vec{x}_n)}{N(\vec{x}_1, \dots, \vec{x}_{n-1})}$$

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

$$\begin{aligned} \nabla_1 F(\theta_1, \dots, \theta_i) \\ \vdots \\ \nabla_n F(\theta_1, \dots, \theta_i) \end{aligned}$$

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

$$(255) \quad (\theta'_1, \dots, \theta'_i) := (\theta_1, \dots, \theta_i) - l \sum_{j=1}^n \nabla_j F(\theta_1, \dots, \theta_i)$$

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

$$(256) \quad l = \frac{1}{n} l'$$

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

18.8 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.

19 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.

19.1 RNN für *sequence labelling*

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

$$(257) \ell : Lex^+ \rightarrow label^+$$

wobei gilt

$$(258) |\ell(\bar{x})| = |\bar{x}|$$

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

$$(vec(w_1), \dots, vec(w_n))$$

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 Eingabeprefix eine Funktion haben:

$$(259) RNN(\vec{x}_1, \dots, \vec{x}_j) : label \rightarrow [0, 1]$$

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

$$(260) RNN_1 : \mathbb{R}^n \rightarrow \mathbb{R}^{|label|}$$

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

$$(261) softmax(x_1, \dots, x_n)_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

softmax liefert eine Wahrscheinlichkeitsverteilung im Sinne, dass für

$$(262) softmax : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

gilt:

$$(263) \sum_{i=1}^n \text{softmax}(\vec{x}) = 1$$

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

$$(264) \text{label} = \{l_1, \dots, l_m\}$$

haben wir

$$(265) P_{RNN}(l_j | \vec{x}, \dots, \vec{x}_n) = (\text{softmax} \circ RNN_1(\vec{x}, \dots, \vec{x}_n))_j$$

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

$$\begin{aligned} &\vec{x}_1, \dots, \vec{x}_i, \\ &1 \leq j \leq i, \end{aligned}$$

setzen wir

$$(266) \hat{j} = \underset{m \in \{1, \dots, |\text{label}|\}}{\text{argmax}} (\text{softmax} \circ RNN_1(\vec{x}_1, \dots, \vec{x}_j))_m$$

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:

$$(267) RNN(\vec{x}_1, \dots, \vec{x}_j) = \text{label}_{\hat{j}}$$

und zuletzt

$$(268) RNN^+(\vec{x}_1, \dots, \vec{x}_i) = RNN(\vec{x}_1) \dots RNN((\vec{x}_1, \dots, \vec{x}_i))$$

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

$$D = \{(\bar{x}_1, \bar{y}_1), \dots, (\bar{x}_m, \bar{y}_m)\}$$

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

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

$$(269) \quad P_{RNN}(-|\vec{x}, \dots, \vec{x}_n) = (softmax \circ RNN_1(\vec{x}, \dots, \vec{x}_n)) : label \rightarrow [0, 1]$$

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

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

gilt zunächst für das Präfix

$$(270) \quad K(x_1, y_1) = -\log P_{RNN}(y_1 | vec(x_1))$$

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:

$$(271) \quad K(x_1, y_1) = -\log P_{RNN}(y_1 | vec(x_1))$$

$$K(x_1, y_1, y_2) = -\log P_{RNN}(y_1 | vec(x_1)) + (-\log P_{RNN}(y_2 | vec(x_1), vec(x_2)))$$

$$(272) \quad \vdots$$

$$(273) \quad \vdots$$

$$(274) \quad K(x_1 \dots x_n, y_1 \dots y_n) = -\sum_{j=1}^n \log P_{RNN}(y_j | vec(x_1), \dots, vec(x_j))$$

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 minimieren, 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.

19.2 RNNs als Sprachmodelle

!!!Bearbeiten

Ein Sprachmodell ist eine diskrete Wahrscheinlichkeitsfunktion

$$(275) P : \Sigma^* \rightarrow [0, 1]$$

Wahrscheinlichkeit ist klar, diskret heißt einfach soviel wie: jedes $w \in \Sigma^*$ bekommt eine Wahrscheinlichkeit $P(w)$ zugewiesen. Sprachmodelle sind wichtige Teilmodule von z.B.

- Übersetzungsmodellen (Qualität der Ausgabe wird bewertet)
- Sprachgenerierungsmodellen (ebenso)
- ...

Also wir brauchen Sprachmodelle immer, wenn Modelle die Qualität von selbst erzeugtem Text kontrollieren müssen.

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

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

eine Wahrscheinlichkeitsverteilung zu induzieren, also:

$$(276) RNN \cong P_{RNN}(-|w_1, \dots, w_{n-1}) : Lex \rightarrow [0, 1]$$

soll eine bedingte Verteilung simulieren. Man macht das über Zustände:

$$(277) N((vec(w_1), vec(w_2), \dots, vec(w_{n-1}), s_0) = s_{n-1})$$

$$(278) P_{RNN}(w_n|w_1, \dots, w_{n-1}) \cong O(N(w_n, s_{n-1}))$$

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 dann

$$P_{RNN}(w_1 \dots w_n) = p(N(vec(w_1), s_0) \cdot p(N(vec(w_2), s_1)) \cdot p(N(vec(w_3), s_2)) \cdot \dots \cdot p(N(vec(w_n), s_{n-1})),$$

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 $w_1 \dots w_{n-1}$ eine einfache Wahrscheinlichkeitsfunktion

$$(279) \quad \hat{P}(w|w_1 \dots w_{n-1}) = \frac{\text{count}(w_1 \dots w_{n-1} w)}{\text{count}(w_1 \dots w_{n-1})}$$

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

$$(280) \quad HK(p \circ N, \hat{P})$$

minimieren, wobei HK die Kreuzentropie zweier Variablen ist. Dementsprechend ist unsere Kostenfunktion am Datenpunkt \bar{w} (ein Satz!)

$$(281) \quad - \sum_{i=1}^{|\bar{w}|} p \circ N(X_i = w_i) \cdot \log(\hat{p}(X_i = w_i))$$

Hier ist X_i eine Zufallsvariable, die dafür steht, wie das i -te Wort im Satz einen gewissen Wert annimmt. Diesen Term versuchen wir zu minimieren!

20 Architekturen mit *gates* – LSTM

20.1 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. Insbesondere wird, durch die Aktivierungsfunktion jeder Zustand in jedem Rechenschritt **normalisiert**, d.h. wieder auf $[0, 1]$ abgebildet. Man kann als in gewissem Sinne *nicht zählen* (sowie Keller-Automaten zählen), da Zahlen keine beliebig großen Werte annehmen können, und z.B.

$$(282) \quad S(10) \approx S(10^{100})$$

Es macht also für den praktisch Gradienten keinen Unterschied!

Architekturen mit *gates* raffinieren und erweitern das Konstruktionsprinzip von RNN. Wie gesagt kann man das ganze automatentheoretisch auffassen: ein RNN ist wie ein Automat; es hat eine Folge von Eingaben

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

und nach jeder Eingabe geht es in einen neuen Zustand \vec{s}_i ; dementsprechend bekommen wir auch eine Zustandsfolge

$$\vec{s}_0 \xrightarrow{\vec{x}_1} \vec{s}_1 \xrightarrow{\vec{x}_2} \dots \xrightarrow{\vec{x}_n} \vec{s}_n$$

Das Problem ist: an jedem Zeitpunkt t_i hat unser Netz nur Zugriff auf den unmittelbar vorhergehenden Zustand und die neue Eingabe. Aber wohlge-merkt: das ist kein Problem für Berechnungen an sich, denn da der Zustandsraum unendlich ist, können wir immer irgendwie 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* (punktweise Multiplikation) 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

$$\mathbf{1} = \overbrace{(1, \dots, 1)}^{k \text{ mal}}$$

Dann können wir ein einfaches **gate** definieren mittels der Definition:

$$(283) \quad \vec{s}_{i+1} = \vec{g} \odot \vec{x}_{i+1} + (\mathbf{1} - \vec{g}) \odot \vec{s}_i$$

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 *a priori* (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:

nimm ein $\vec{g}' \in \mathbb{R}^k$ 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 z.B.

$$S(\vec{g}') \odot \vec{x}$$

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.

20.2 LSTM – Definition

Die *long-short-term-memory* Architektur von Hochreiter& Schmidhuber 1997 adressiert 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:

$$(284) \quad \vec{s}_{j+1} = N_{LSTM}(\vec{s}_j, \vec{x}_{j+1}) = (\vec{c}_{j+1}, \vec{h}_{j+1})$$

wobei

$$(285) \quad \vec{c}_{j+1} = \vec{f} \odot \vec{c}_j + \vec{i} \odot \vec{z}$$

$$(286) \quad \vec{h}_{j+1} = \vec{o} \odot \tanh(\vec{c}_{j+1})$$

$$(287) \quad \vec{i} = S(M^{xi} \vec{x}_{j+1} + M^{hi} \vec{h}_j)$$

$$(288) \quad \vec{f} = S(M^{xf} \vec{x}_{j+1} + M^{hf} \vec{h}_j)$$

$$(289) \quad \vec{o} = S(M^{xo} \vec{x}_{j+1} + M^{ho} \vec{h}_j)$$

$$(290) \quad \vec{z} = \tanh(M^{xz} \vec{x}_{j+1} + M^{hz} \vec{h}_j)$$

Außerdem wird die Ausgabe definiert als

$$(291) \quad \vec{y}_{j+1} = O_{LSTM}(\vec{s}_j) = \vec{h}_j$$

das bedeutet: der Arbeitsspeicher funktioniert als Ausgabe. Unser Zustand ist also erstmal zweigeteilt; gleichzeitig gibt es 3 gates, nämlich

1. **input**,
2. **forget**
3. **output**

Alle drei gates berechnen ein Perzeptron, also eine lineare Funktion \circ 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 Arbeitsspeicher 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-Kandidat \vec{z} in $[-1, 1]$ liegen. Daraus folgt:
- der Langzeit-Speicher \vec{c} ist der *einzig*e 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 (285) 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).

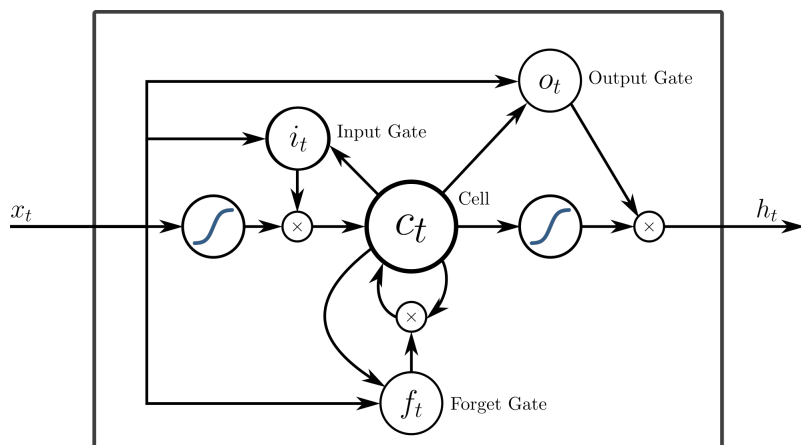
Wichtig ist es, zu verstehen was die Parameter des Netzes sind:

- 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, oder wahlweise beliebig große/kleine Werte annimmt.



LSTM liefern derzeit state-of-the-art Ergebnisse für die meisten Aufgaben, die das Modellieren von Sequenzen erfordern. Ein Problem dieser Architektur ist, dass 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).

20.3 Gated recurrent units

GRU basieren auf denselben Ideen von gates, haben aber eine einfachere Architektur und keine separaten Speicherabteile. Sie sind wie folgt definiert:

$$(292) \quad \vec{h}_{j+1} = N_{GRU}(\vec{h}_j, \vec{x}_{j+1}) = (\mathbf{1} - \vec{z}) \odot \vec{h}_j + \vec{z} \odot \tilde{h}_{j+1}$$

wobei

$$(293) \quad \vec{z} = S(M^{xz}\vec{x}_{j+1} + M^{hz}\vec{h}_j)$$

$$(294) \quad \vec{r} = S(M^{xr}\vec{x}_{j+1} + M^{hr}\vec{h}_j)$$

$$(295) \quad \tilde{h}_{j+1} = \tanh(M^{xh}\vec{x}_{j+1} + M^{hh}(\vec{h}_j \odot \vec{r}))$$

Der output ist gegeben durch

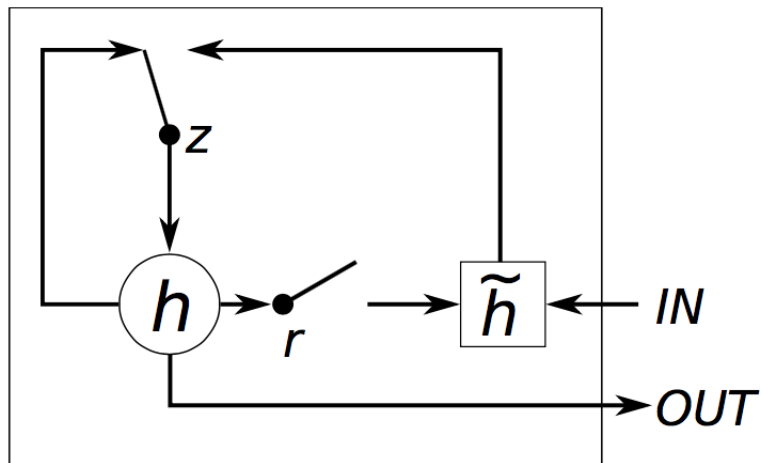
$$(296) \quad \vec{y}_{j+1} = O_{GRU}(\vec{h}_{j+1}) = \tilde{h}_{j+1}$$

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.



21 Convolutional Neural Networks

21.1 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.

21.2 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** berücksichtigen als vergangene Messungen, anders gesagt: es gibt eine Funktion

$$(297) \quad w : \mathbb{R} \rightarrow [0, 1]$$

die *monoton fällt*, die uns sagt wie wir unsere Messung berücksichtigen sollen. Diese Funktion nimmt also als Argument die Zeitspanne, die eine Messung in der Vergangenheit liegt, und liefert als Ausgabe deren “Gewicht” für die aktuelle Messung.

Unser Sensor dagegen liefert eine Funktion

$$(298) \quad s : \mathbb{R} \rightarrow \mathbb{R}^n$$

wobei das Argument der Zeitpunkt der Messung ist, und der Wert das Messergebnis (eine Koordinate im Raum). Wenn nun ein Messergebnis zum Zeitpunkt a gemessen wurde, dann wird es gewichtet (am Zeitpunkt $t > a$) mit

$$(299) \quad w(t - a)$$

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

$$(300) \quad \mu(t) = \int_{0 < a < t} s(a)w(t-a)da$$

wobei a die Zeitvariable über die gesamten vergangenen Messungen ist, $w(t-a)$ das gegenwärtig (an t) darauf gelegte Gewicht. Wir integrieren also über das Alter der Messung. In diesem Fall muss w eine Wahrscheinlichkeitsdichtefunktion sein, die darüberhinaus für negative Werte 0 liefert, also:

$$(301) \quad \int_{-\infty}^{\infty} w(x)dx = 1$$

$$(302) \quad x < 0 \implies w(x) = 0 :$$

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:

$$(303) \quad \mu(t) = \sum_{a=0}^t s(a)w(t-a)$$

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:

$$(304) \quad \mu(t) = (s * w)(t)$$

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:

$$(305) \quad S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$

Das erste wichtige Ergebnis ist, dass Konvolution **kommutativ** ist:

$$(306) \quad (I * K)(i, j) = (K * I)(i, j)$$

was explizit bedeutet:

$$(307) \quad \sum_m \sum_n I(m, n)K(i - m, j - n) = \sum_m \sum_n I(i - m, j - n)K(m, n)$$

In der Praxis benutzt man oft eine etwas andere Funktion, die man **cross-correlation** nennt, und die wie folgt definiert ist:

$$(308) \quad S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

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 am Mittelwert der bisherigen Messung sein.

Vorhersage 2 Die folgende Messung wird relativ nah an der letzten 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.

21.3 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).

(309) 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

$$vec : Lex \rightarrow \mathbb{R}^L$$

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

$$(\#) \vec{x}_1, \dots, \vec{x}_n$$

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

- $\vec{x}_1, \dots, \vec{x}_k$
- $\vec{x}_2, \dots, \vec{x}_{k+1}$
- ...
- $\vec{x}_{(n-k)+1}, \dots, \vec{x}_n$

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 \oplus mittels

$$(310) \quad \oplus(\vec{x}_1, \dots, \vec{x}_k) = (x_1^1, \dots, x_1^L, \dots, x_k^1, \dots, x_k^L)$$

wir schreiben also, in Worten, die Komponenten aller Vektoren in einen neuen Vektor der Grösse

$$\oplus(\vec{x}_1, \dots, \vec{x}_k) \in \mathbb{R}^{L \cdot k}$$

Einfachheit halber setzen wir

$$(311) \quad \vec{y}_i = \oplus(\vec{x}_i, \dots, \vec{x}_{i+k})$$

Dieser Vektor enthält, in einem einfachen Sinne, die Information des k -grams von Worten, und die Operation \oplus brauchen wir nur, um möglichst einfach darauf zugreifen zu können. Nun wenden wir auf diese Vektoren ein einfaches Perzeptron an, also eine Funktion

$$(312) \quad g \circ M : \mathbb{R}^{L \cdot k} \rightarrow \mathbb{R}^i$$

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

$$(313) \quad \vec{p}_i = g(M\vec{y}_i + \vec{a})$$

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 k -grams kodieren.

- ▷ Für die k -gramme selber spielt die Reihenfolge der Worte durchaus eine Rolle;
- ▷ Die Reihenfolge der k -gramme 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 (in diesem Fall: $n - (k - 1)$) 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

$$pool : (\mathbb{R}^j)^n \rightarrow \mathbb{R}^i$$

(wichtig: n variabel, j und i fix!) 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:

$$(314) \quad mp : (\mathbb{R}^j)^n \rightarrow \mathbb{R}^j$$

was definiert ist durch die *max*-Operation:

$$(315) \quad (mp(\vec{y}_1, \dots, \vec{y}_n))_m = \max_{1 \leq i \leq n} (y_1^m, \dots, y_n^m)$$

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:

$$(316) \quad (ap(\vec{y}_1, \dots, \vec{y}_n))_m = \frac{1}{n} \sum_{l=1}^n (\vec{y}_l)_m$$

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.

$$(317) \quad \begin{pmatrix} 1 & 5 & 8 \\ 6 & 9 & 0 \\ 4 & 6 & 1 \\ 9 & 2 & 4 \\ 1 & 7 & 3 \\ 0 & 3 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 9 & 8 \\ 6 & 7 & 7 \end{pmatrix}$$

wenn $k = 2$. Nun haben wir allerdings noch keinen Vektor, sondern eine Matrix. Wir machen daraus einen Vektor, indem wir die Vektor-komponenten konkatenieren:

$$(318) \begin{pmatrix} 9 & 9 & 8 \\ 6 & 7 & 7 \end{pmatrix} \Rightarrow (9 \ 9 \ 8 \ 6 \ 7 \ 7)$$

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.

21.4 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 einen 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.

22 Kostenfunktionen – intern

Es gibt zwei Arten von Kostenfunktionen:

1. Solche, mit denen wir unsere Modelle evaluieren, vergleichen etc. Man kann sie **extern** nennen, denn sie spielen für das Modell selber keine Rolle, sondern nur für den Menschen.
2. Solche, deren Werte (also Kosten) das Modell selbst im Training zu minimieren sucht. Diese Kostenfunktionen sind ein integraler Part des Modells; eine andere Kostenfunktion führt zu einem anderen Trainingsergebnis. Diese Kosten sind also **intern** und spielen erstmal zur Evaluation keine große Rolle.

Das schließt natürlich nicht aus, dass interne und externe Kosten manchmal zusammenfallen. Hier interessieren uns in erster Linie interne Kostenfunktionen, und dabei in erster Linie Kostenfunktionen für Klassifikatoren.

Externe Kostenfunktionen basieren zumeist auf der akkuraten Klassifizierung von Daten. Das ganze wird dann normalisiert, oder nochmal verfeinert durch Unterscheidung verschiedener Arten von Fehlern (precision oder recall, zusammen im F1-score).

Das Problem solcher Metriken ist grundsätzlich folgendes: sie sind **eckig**, also nicht differenzierbar. Minimale Änderungen der Parameter werden grundsätzlich nichts am score ändern, bevor er sprunghaft steigt. Das bedeutet: nicht differenzierbar, nicht konvex, also ist eine (automatische) Optimierung grundsätzlich kaum möglich.

Wir haben also folgendes grundsätzliche Problem: wir suchen eine stetige, differenzierbare Funktion um ein diskretes Problem zu lösen. Die Lösung unseres Problems ist folgende: wir lassen unsere Klassifikatoren ja eine Wahrscheinlichkeitsverteilung ausgeben, auf deren Basis wir klassifizieren.

$$(319) \quad N(\vec{x}) = \text{softmax}(N'(\vec{x})) = (p_1, \dots, p_n)$$

wobei

$$(320) \quad p_i = P(l_i|\vec{x})$$

und die Klassifikation ist gegeben durch

$$(321) \operatorname{argmax}_i P(l_i | \vec{x})$$

Im Prinzip kann man auch die “richtige Klassifikation” auch als eine Wahrscheinlichkeitsverteilung auffassen, eine besondere

$$(0, \dots, 0, 1, 0, \dots, 0)$$

Wir lassen nun den letzten Schritt und arbeiten nur mit den Wahrscheinlichkeiten des Modells. Wir nutzen die probabilistischen Begriffe und den Begriff der Entropie, um eine stetig differenzierbare Kostenfunktion zu definieren. Insbesondere möchten wir die Kreuzentropie zwischen den beiden Verteilungen minimieren.

22.1 Entropie etc.

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

$$(322) \quad H_P(\omega) := P(\omega) \cdot -\log(P(\omega))$$

Die Entropie eines einzelnen Ereignisses ist normalerweise weniger interessant als die Entropie einer ganzen Verteilung P (über einen diskreten Raum Ω , geschrieben $H(P)$):

$$(323) \quad H(P) := - \sum_{\omega \in \Omega} P(\omega) \log(P(\omega))$$

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(i in 1 : 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

$$\text{maximale Entropie} \approx \text{maximale Unwissenheit}$$

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:

$$(324) \quad H(X) := - \sum_{x \in X} P(X = x) \log(P(X = x))$$

22.2 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):

$$(325) \quad H(X|Y) = \sum_{y \in Y} P(Y = y) H(X|Y = y)$$

Wenn wir diese Definition auflösen, bekommen wir:

$$(326) \quad H(X|Y) = \sum_{x \in X, y \in Y} P(X^{-1}(x) \cap Y^{-1}(y)) \log \left(\frac{P(X^{-1}(x) \cap Y^{-1}(y))}{P(Y^{-1}(y))} \right)$$

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

$$(327) \quad H(X|Y) = 0$$

insbesondere also:

$$(328) \quad H(X|X) = 0$$

Umgekehrt, falls der Wert von Y keinerlei Einfluss hat auf die Wahrscheinlichkeitsverteilung des Wertes von X , dann haben wir

$$(329) \quad H(X|Y) = H(X)$$

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:

$$(330) \quad H(X|Y) = H(\langle X, Y \rangle) - H(Y)$$

wobei $\langle X, Y \rangle$ eine neue Variable ist, mit

$$(331) \quad P(\langle X, Y \rangle = (x, y)) = P(X = x, Y = y) = P(X^{-1}(x) \cap Y^{-1}(y))$$

Wir nehmen also die Entropie der **Verbundverteilung**, und ziehen die Entropie von $H(Y)$ ab.

22.3 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:

$$(332) \quad D_{KL}(P\|Q) = \sum_{\omega \in \Omega} P(\omega) \log \frac{P(\omega)}{Q(\omega)}$$

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 $H(X|Y)$, das 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, das wir geschätzt haben.

22.4 Kreuzentropie

Die Definition der Kreuzentropie ist nun

$$(333) \quad HK(X, P, Q) = H(X) + D(P||Q)$$

Äquivalent dazu gibt es (im diskreten Wahrscheinlichkeitsraum, den wir hier betrachten werden).

$$(334) \quad HK(X, P, Q) = - \sum_{x \in \Omega} P(X = x) \cdot \log(Q(X = x))$$

Man beachte: im Anwendungsfall ist P ist “korrekte Verteilung”, dargestellt als ein Vektor $\vec{y} = (0, \dots, 0, 1, 0, \dots, 0)$ mit einer 1, sonst Nullen (Einheitsvektor). Q dagegen ist z.B. die Ausgabe der *softmax*-Funktion, als Vektor geschrieben $\vec{\hat{y}}$. Man sieht das 334 sich reduziert zu

$$(335) \quad -\log(\vec{y}_i), \text{ wobei } \vec{y}_i = 1$$

Denn alle anderen Terme reduzieren sich auf 0. Man kann das übrigens auch als reine Vektor-Multiplikation schreiben:

$$(336) \quad HK(\vec{y}, \vec{\hat{y}}) = -(\vec{y}^\top \log \vec{\hat{y}})$$

wobei \log komponentenweise appliziert wird. Übrigens spielt für diesen Wert erstmal *jeder* Parameter eine Rolle (entgegen dem Anschein), denn in der *softmax*-Funktion kommen alle Komponenten zusammen!

Diese Darstellung hat einen weiteren Vorteil: man kann mit ihr gewisse Fehler weniger stark oder stärker Gewichten als andere. Z.B.: wir möchten Fehler bei Klasse 1 sagen wir doppelt gewichten zu Fehler der Klasse 2. D.h. unsere \vec{y} haben die Form

$$(2, 0, 0, \dots), \text{ aber } (0, 1, 0, 0, \dots)$$

Diesen Ansatz kann man natürlich beliebig variieren.

22.5 Kostenfunktion für binäre Klassifizierung

Wie berechnen wir die Kosten von einer gegebenen Klassifizierungsfunktion, so dass wir die Parameter des linearen Teils optimieren können? Hier nimmt man die sog. log-loss Funktion, die jeweils die logarithmische Wahrscheinlichkeit des Falsch-liegens misst: Zur Erinnerung:

- Y ist eine Zufallsvariable, die den Wert 0 oder 1 annehmen kann, jeweils mit einer gewissen Wahrscheinlichkeit. Dabei ist diese Wahrscheinlichkeit durch die Funktion gegeben:

$$(337) \quad P_\theta(Y = 1|\vec{x}) = S(f_\theta(\vec{x}))$$

f_θ die lineare Funktion mit θ als Parametern.

- Mit y , als “korrekter Datenpunkt”, ist in $\{0, 1\}$.
- $m = |D|$ ist die Größe des Datensatzes.

Dann bekommen wir folgende Funktion:

Log-loss

$$(338) \quad K(\theta) = -\frac{1}{m} \sum_{(\vec{x}, y) \in D} [y \cdot \log P_\theta(Y = 1|\vec{x}) + (1 - y) \cdot \log(1 - P(Y = 1))]$$

Dieser Log-loss ist tatsächlich nichts anderes als die Kreuzentropie, die wir oben definiert haben!

Wir nehmen also die Summe der beiden Abweichungen vom richtigen Wert, jeweils in beide Richtungen. Wenn wir genau richtig liegen würden, wäre der Term 0 bzw. undefiniert ($\log 0 = -\infty$); da $S(x) \in (0, 1)$ kann das aber nicht passieren. Allgemein gilt: je näher $P(Y = y|\vec{x})$ an y liegt, desto kleiner ist der Term.

Wir sehen also hier, dass probabilistische Konzepte eine entscheidende Rolle für das Training spielen. Ohne sie gibt es normalerweise keine Möglichkeit, Klassifizierung effektiv zu optimieren.

22.6 Kostenfunktion für beliebige Klassifizierung

Auch hier kann man *cross-entropy* nutzen, allerdings löst sich die Gleichung nicht so schön und einfach auf. Wir können jede Ausgabe in eine Wahrscheinlichkeitsverteilung umwandeln, indem man eine softmax-Funktion nachschaltet:

$$(339) \text{ softmax} \circ N(\vec{x}) = P_N : L \rightarrow [0, 1]$$

das bedeutet, die Ausgabe des Netzes ist eine Verteilung über die label. Gleichzeitig haben wir eine "korrekte" Datenfunktion

$$(340) D : \vec{x} \rightarrow L$$

(L die Menge der labels) welche somit jeder Eingabe das korrekte label zuweist. Die Kostenfunktion reduziert sich also auf

$$(341) K(N, \vec{x}) = -\log(P_N(D(\vec{x})))$$

Einfach gesagt: man nimmt den negativen Logarithmus der Wahrscheinlichkeit, den unser Netz der richtigen Komponente zuweist. Diesen Term kann man sehr einfach optimieren, indem man die Wahrscheinlichkeit $P_N(D(\vec{x}))$ erhöht.

23 Sequence-to-sequence Modelle

23.1 Einfache Modelle

Wichtig ist folgendes: das Modell der Sequenzenverarbeitung ist unabhängig vom Modell der Eingabenverarbeitung. Alle hier besprochenen Modelle, sofern nicht explizit anders gesagt, funktionieren gleichermaßen mit RNN, LSTM, GRU und was auch immer.

Wir haben bereits zwei Modelle der Sequenzenverarbeitung besprochen:

1. Eingabesequenz zu Ausgabesequenz gleicher Länge $\vec{x}^1, \dots, \vec{x}^n \implies \vec{y}^1, \dots, \vec{y}^n$, wie in POS-tagging
2. Eingabesequenz zu Vektor: $\vec{x}^1, \dots, \vec{x}^n \implies \vec{y}$, wie in sentiment detection, Bilderkennung etc.

Diese beiden Modelle der Sequenzenverarbeitung gibt es immer; im ersten Fall berechnen wir nach jeder Eingabe eine Ausgabe, im zweiten Fall machen wir im Prinzip dasselbe, nur dass uns nur die letzte Ausgabe interessiert.

23.2 Bidirektionale Modelle

Ein grundsätzliches Problem von neuronalen Netzen in der Sequenzenverarbeitung ist, dass sie asymmetrisch sind: die bereits gelesenen Eingaben haben einen Einfluss auf den aktuellen Zustand, die noch zu lesenden Eingaben dagegen nicht.

Wir erinnern uns an etwas ähnliches bei Markov Modellen: aber hier wird die scheinbare Asymmetrie durch die Gesetze der Wahrscheinlichkeitstheorie wieder **“ausgeglichen”**.

Neuronale Netze sind relativ wenig verstanden, es gibt aber keine Evidenz dafür dass es eine vergleichbare Symmetrie gibt. Dafür gibt es starke empirische Evidenz dafür, dass bidirektionale Modelle besser abschneiden bei vielen Aufgaben.

Wir definieren $[-]^T$ als die Transposition von Sequenzen; sprich:

$$(342) \quad (\vec{x}^1, \dots, \vec{x}^n)^T = (\vec{x}^n, \dots, \vec{x}^1)$$

Insbesondere bleiben die Zahlenvektoren selbst aber unverändert, also

$$(343) \quad (\vec{x})^T = \vec{x}$$

Die Idee des bidirektionalen Netzes ist:

- Wir nehmen eine Eingabesequenz $(\vec{x}^1, \dots, \vec{x}^n)$ und applizieren darauf *zwei* neuronale Netze.
 1. eines auf $(\vec{x}^1, \dots, \vec{x}^n)$ selber,
 2. eines auf $(\vec{x}^1, \dots, \vec{x}^n)^T$
- Die beiden Netze operieren unabhängig voneinander; erst im Ausgabebayer werden die Berechnungen der beiden Netze zusammengefügt.
- Wie genau das Training abläuft weiß ich nicht. Ich glaube aber unabhängig voneinander auf D bzw. D^T .

Lange Zeit (~ 2 Jahre) war das Bi-LSTM der letzte Schrei.

Es bleibt hier aber die Beschränkung: Eingabe und Ausgabe haben dieselbe Länge, oder die Ausgabe hat Länge 1.

23.3 Vector to sequence (VtS)

(Nach Goodfellow, Bengio, Courville) Wir haben besprochen, wie man Sequenzen zu einzelnen Vektoren abbildet; das ist die typische Aufgabe in der Klassifikation von Sequenzen. Es gibt auch das Gegenstück: aus einem einzelnen Vektor (fixe Länge) soll eine Sequenz von Vektoren generiert werden. Hierfür gibt es weit weniger Anwendungen; eine mir bekannte ist **Bildbeschriftung**: ein Bild (festgelegte Pixel \cong festgelegte Vektorgröße) soll mit Text beschrieben werden (also variable Länge). Allgemeiner kann man sagen: VtS als Modell ist interessant v.a. in der **Sprachgenerierung**, wenn wir eine Information versprachlichen möchten.

VtS Modelle weichen deutlich ab von anderen Modellen die wir besprochen haben. Das auffälligste an Ihnen ist:

- Die (eine) Eingabe wird in jedem Zustand neu verwertet.
- Der Zustand selbst entscheidet, ob die Berechnung terminiert oder nicht.

Wir haben also folgende Komponenten (im Falle des einfachen Perzeptrons):

1. Die Eingabe \vec{x}
2. Eine Matrix R , welche die Eingabe \vec{x} *immer gleich* vorverarbeitet. R ist also invariant sowohl über Eingaben als auch über Zeitschritte.
3. Ein Zustand h_n , der als Eingabe nimmt $R\vec{x}$ und y_n (die aktuelle Ausgabe)
4. Eine Ausgavevektor o , Funktion von h , welcher eine Wahrscheinlichkeitsverteilung über Ausgabe y_{n+1} darstellt (also Nachfolger!)

Wir haben also in diesem Fall folgende Gleichungen:

$$(344) \quad \vec{h}_{j+1} = g(R\vec{x} + U\vec{y}_{j+1} + W\vec{h}_j)$$

$$(345) \quad \vec{o}_{j+1} = \text{softmax}(V\vec{h}_j)$$

$$(346) \quad \vec{L}_j = -\log(\pi_{y_{j+1}}(\vec{o}_j))$$

Hierbei muss man erklären: \vec{o} wird gelesen als $P : Y \rightarrow [0, 1]$, wobei jeder $P(y) = \pi_y(\vec{o})$, also jede Projektion entspricht einer Wahrscheinlichkeit.

Was ist also die Ausgabe des Netzes $N(\vec{x})$? Das ist nicht ganz einfach, denn man muss das ganze **Dekodieren** (man kennt das Prinzip aus der maschinellen Übersetzung): wir fassen L_j als die jeweilige Ausgabe des Netzes aus, also

$$(347) \quad N(\vec{x}, y_1, \dots, y_n) = L_1, \dots, L_n$$

Dann suchen wir:

$$(348) \quad \underset{\vec{y} \in Y^*}{\operatorname{argmax}} \sum N(\vec{x}, y_1, \dots, y_n)$$

gegeben die Eingabe \vec{x} . Wir können das als eine Wahrscheinlichkeit auffassen, nämlich

$$(349) \quad \underset{\vec{y} \in Y^*}{\operatorname{argmax}} \sum N(\vec{x}, y_1, \dots, y_n) \cong -\log P(y_1, \dots, y_n | \vec{x})$$

Man beachte dass jedes y eigentlich in label ist, es wird aber repräsentiert durch einen Vektor!

Wir suchen also die maximal wahrscheinliche Ausgabe. Das ist nicht trivial: da alle Ausgaben voneinander abhängen (Ausgabe beeinflusst Zustand!), reicht es nicht, lokal das Maximum zu suchen! Man nennt diese Aufgabe (argmax suchen) **Dekodierung** des Modells.

(Es fehlt die Bedingung für den Ausgabestop – kann ich nichts zu sagen!)

23.4 Encoder – decoder (ED)

Wir haben nun beide Richtungen beschrieben:

- Sequenz von Vektoren \implies Vektor
- Vektor \implies Sequenz von Vektoren

Das ED Modell arbeitet nun, indem einfach die beiden Abbildungen **komponiert** werden:

Eingabe Sequenz \implies Vektor \implies Ausgabesequenz

Folgendes:

- Es ist natürlich klar dass dieses Modell sehr viele Trainingsdaten benötigt
- Es ist ebenso klar, dass Eingabe und Ausgabelänge unabhängig sind
- Man kann hier, im Falle von maschineller Übersetzung, fast von einer semantischen Dimension sprechen: der “Mittelvektor” soll den Inhalt repräsentieren

Neural machine translation arbeitete für eine beträchtliche Zeit (vielleicht 2 Jahre) nach dem ED-Modell. Das größte Problem, soweit ich das verstehe, war folgendes:

- Der Mittelvektor muss den gesamten Inhalt “speichern”;
- da aber seine Größe konstant ist, kann das bei langen Eingaben sehr schwierig bis praktisch unmöglich werden.

Eine Lösung hierfür war: anstatt den gesamten Satz auf einmal zu übersetzen, sucht man sich Portionen aus (also Teilsätze), die dann nach dem ED-Modell übersetzt werden. Das funktioniert gut für Sprachen, die auf der “Makro-Ebene” parallel aliniert sind (Englisch-Französisch), aber schlecht für Sprachen in das nicht gegeben ist (Englisch-Japanisch).

Diese Begrenzungen führen uns zur **Attention**, was jetzt der letzte Schrei ist (u.a. im neural maschine translation).

24 Attention

24.1 Intuition: Alinierung und Wahrscheinlichkeit

Streng genommen ist Attention kein Seq2seq model: die Eingabelänge ist *konstant*. Man nutzt Attention aber als Seq2seq model, indem man

- trainiert mit einer gegebenen festen Länge, die groß genug ist (üblich: 512 Worte). Alle Eingaben werden dann auf diese Länge *gepadded* (ein Symbol füllt die offenen Stellen auf)
- alle späteren Eingaben, die länger sind, einfach abschneidet (truncate).

Das ist nicht wirklich elegant, funktioniert aber i.A. sehr gut, da Sätze sehr selten sehr lang sind.

Attention ist also auch eine Methode der Sequence-to-Sequence Verarbeitung. Eingeführt wurde das Prinzip in der maschinellen Übersetzung, wo es auch am wichtigsten ist. Wir rekapitulieren das Problem der Neural Machine Translation soweit:

Alle Informationen des Eingabesatzes (beliebig lang) müssen in einen Vektor konstanter Länge kodiert werden.

Das führt insbesondere zu Problemen, wenn wir Sätze übersetzen wollen, die länger sind als die meisten unserer Trainingsdaten.

Der “traditionelle” Ansatz in statistischer maschineller Übersetzung ist folgender: wir haben eine Alinierung zwischen Eingabe und Ausgabe, genauer gesagt eine Wahrscheinlichkeitsverteilung über Alinierungen. Um die Übersetzungswahrscheinlichkeit von Worten zu bestimmen, gewichten wir sie mit der Alinierungswahrscheinlichkeit. Die Ausgabe wird dabei von links nach rechts konstruiert, aber jedes Wort kann dabei mit einem beliebigen Wort der Eingabe aliniert sein.

Wir möchten dieses Konzept in die NMT übertragen. Das Prinzip ist folgendes:

- Die Ausgabe wird linear von links nach rechts konstruiert, aber
- die gesamte Eingabe steht in jedem Schritt vollständig zur Verfügung, aber
- unsere Aufmerksamkeit richtet sich in jedem Schritt auf einen anderen Teil der Eingabe

- Aufmerksamkeit ist dabei eine Funktion der bisherigen Berechnungen (innerer Zustand) und der bisherigen Ausgabe

Was aber ist Aufmerksamkeit?

Definition 12 *Aufmerksamkeit über eine Eingabe $(\vec{x}^1, \dots, \vec{x}^n)$ ist eine gewichtete Summe $\sum_{i=1}^n \lambda_i \vec{x}^i$, wobei λ_i ein Skalar, und $\sum_{i=1}^n \lambda_i = 1$.*

Wir repräsentieren also eine Aufmerksamkeit über eine Eingabesequenz der Länge n mit einem Vektor $(\lambda_1, \dots, \lambda_n)$, wobei

$$(350) \quad att[\lambda_1, \dots, \lambda_n](\vec{x}^1, \dots, \vec{x}^n) = \sum_{i=1}^n \lambda_i \vec{x}^i$$

Dieses Konzept, auch wenn es erstmal seltsam klingt, ist durchaus einleuchtend: ein Spezialfall von Aufmerksamkeit wäre die “one-hot”-Variante, sprich

$$(351) \quad att[0, \dots, 1, \dots, 0]$$

in welchem Fall wir uns exklusiv auf ein Wort konzentrieren. Das würde also, im SMT, einer fixen Alinierung entsprechen. Das hat man normalerweise nicht, denn die Aufmerksamkeit soll direkt eine Wahrscheinlichkeitsverteilung über alle Alinierungen darstellen.

Wir nehmen also folgendes an:

- In jedem Rechenschritt des Netzes wird eine Ausgabe(verteilung) \vec{y}_j generiert.
- Die Eingabe ist unter anderem die attention $att_j = (\lambda_1^j, \dots, \lambda_n^j)$

Nun gibt es folgende Korrespondenz:

$$(352) \quad \lambda_i^j \cong P(o_j \text{ ist aliniert mit } in_i)$$

wobei o_j die j te Ausgabe ist, in_i die i te Eingabe.

24.2 Definition - Die erste Architektur

Ich beschreibe hier die Architektur von Bahdanau, Cho, Bengio 2016: Neural Machine Translation by Jointly Learning to Align and Translate.

Die Eingabe Die Eingabe ist wie immer eine Sequenz von Vektoren

$$(\vec{x}^1, \dots, \vec{x}^n),$$

wobei jeder Vektor einem Wort entspricht und das Ergebnis ist eines word-embedding.

Die “Annotation” Die Eingabe wird zunächst bearbeitet von einem Bi-LSTM, sequence-to-sequence. Wir bekommen also für jedes \vec{x}^i zwei Zustände:

1. \vec{h}_i , den i ten Zustand des Forward-Netzes
2. \overleftarrow{h}_i , den i ten Zustand des Backward-Netzes

Die Autoren nennen das die Annotation, denn so stehen in jedem Zustand relevante Informationen über die Vorgänger (\vec{h}_i) und die Nachfolger (\overleftarrow{h}_i). Die Annotation ist nun einfach die Paarung dieser beiden Zustände:

$$(353) \quad ann_i = (\vec{h}_i, \overleftarrow{h}_i)$$

Der Grund für diese Annotation ist folgende: die Eingabe soll die Aufmerksamkeit *steuern*, das geht aber nur, wenn globale Information über links und rechts stehende Worte, sofern relevant, lokal verfügbar ist.

Das Alinierungsmodell Die Alinierung wird errechnet als ein einfaches Perzeptron (glaube ich), appliziert auf den vorigen *internen* Zustand eines neuen Netzes, und die Annotation. Wir schreiben also:

$$(354) \quad e_{ij} = g(\vec{m}^\top(s_{i-1}, ann_j))$$

$$(355) \quad e_{ij} \in \mathbb{R}$$

Wir berechnen also in Schritt i immer n verschiedene Alinierungen, nämlich für jede Annotation eine. Das Ergebnis ist jeweils eine reelle Zahl.

An diesem Punkt ist entscheidend: für diesen Rechenschritt ist die Länge der Eingabe die entscheidende Variable. Denn alle Komponenten unseres Modells haben eine feste Dimensionalität, aber die Anzahl der Gewichte λ_j für die Aufmerksamkeit ist *variabel*. Dieses Problem wird in diesem Rechenschritt gelöst.

Die Aufmerksamkeit selbst wird errechnet nach einem ganz einfachen Muster:

$$(356) \text{ att}_i = \text{softmax}(e_{i1}, \dots, e_{in})$$

Wir definieren $\text{att}_{ij} = (\text{att}_i)_j$, das wäre also die oben beschriebene Wahrscheinlichkeit; also:

$$(357) \text{ att}_{ij} \equiv \lambda_j^i$$

Der Kontext ist im VtS-Modell der eine Eingabevektor, der immer wieder neu eingegeben wird. Hier passiert nun dasselbe: die gesamte Eingabesequenz wird immer wieder, in jedem Schritt, neu gelesen, nur: es ändert sich unsere Aufmerksamkeit!

$$(358) c_i = \sum_{j=1}^n \lambda_j^i \text{ ann}_j$$

Der Kontext ist also das Ergebnis der Aufmerksamkeit, appliziert auf die Annotation – eine gewichtete Summe.

Der Zustand Wir haben bereits den vorigen Zustand s_{i-1} benutzt, um das Alinierungsmodell und damit die Aufmerksamkeit zu errechnen. Wir zeigen nun, wie man den Nachfolgezustand errechnet:

$$(359) s_i = f(s_{i-1}, y_{i-1}, c_i)$$

(Laut meiner Quelle) Hierbei ist f wohl ein einfaches Perzeptron, sprich:

$$(360) s_i = g(M(s_{i-1}, y_{i-1}, c_i))$$

Bleibt noch die Ausgabe y , die wir noch nicht besprochen haben:

Die Ausgabe Hier schweigt das Papier, ich schätze:

$$(361) y_i = \text{softmax}(N(s_i))$$

Damit können wir die Wahrscheinlichkeit jeder Ausgabe o ablesen, nämlich (wie gehabt)

$$(362) L_i(o) = -\log(\pi_o(y_i))$$

Es ist wichtig zu verstehen, dass hier wiederum y_i als Eingabe dient für s_{i+1} – dementsprechend können die verschiedenen Ausgaben sich über die Zeitschritte nach vorne beeinflussen!

Und das ist die erste Architektur mit attention. Man kann sie vereinfacht als eine 4-Ebenen-Architektur darstellen:

1. Die Annotationsebene, ein Bi-LSTM auf den Eingabevektoren
2. Die Ebene der Attention, die aus der Eingabesequenz einen Vektor fester Länge macht
3. Die Zustandsebene s_j , die in jedem Rechenschritt einen Zustand hat
4. Die Ausgabeebene der y_i , die jeweils eine Wahrscheinlichkeitsverteilung über Ausgabeworte repräsentieren

Wir sehen: die Attention ist hier nur eine Ebene in der größeren Architektur, und das ist, vor dem Hintergrund der maschinellen Übersetzung durchaus sinnvoll, denn die Aufmerksamkeit entspricht der Alinierung im SMT.

24.3 Training

Das Training läuft uniform ab, sprich: für ein Eingabe-Ausgabe-Paar werden alle Komponenten zeitgleich optimiert, um die likelihood der Ausgabe gegeben die Eingabe zu maximieren.

24.4 Attention is all you need – der Transformer

Nach dem gleichnamigen Artikel (zu viele Autoren, ich nenne die die Google-Leute). Die Autoren lösen das Konzept der Aufmerksamkeit aus dem ursprünglichen Kontext und konstruieren eine Architektur, die ausschließlich aus attention-layern besteht. Wie geht das? Erstmal konzeptuell:

Attention ist ja nicht rekurrent. Der Transformer besteht nur aus Attention-layern – wie können wir hier beliebige Eingaben und Ausgaben konstruieren? Es gibt also keinen Zustand, der die bisherigen Berechnungen kodiert!

- Die Ausgabe wird wie immer Schritt für Schritt konstruiert
- Die Eingabe wäre an für sich kommutativ: wir gehen ja in keiner Weise sequentiell durch, sondern betrachten immer eine gewichtete Summe
- Die lineare Ordnung der Eingabe(vektoren) wird separat kodiert mit einem Position-encoding. Auf diese Weise ist die Reihenfolge auch in der gewichteten Summe präsent!

Erstmal haben die Google-Leute einen etwas anderen und allgemeineren Begriff von Attention, sie nutzen die sog. *Dot-Product Attention*.

$$(363) \text{ att}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Zunächst sind alle Matrizen hier quadratisch. Weiterhin ist

- Q eine Matrix von Queries (jeder Vektor eine query)
- K ist eine Matrix von Keys (jeder Vektor ein key)
- V ist eine Matrix von Values (jeder Vektor ein value)
- d_k ist die Anzahl der Dimensionen der (einzelnen) keys, die den Term normalisieren soll

Der (normalisierende) Term $\sqrt{d_k}$ ist nicht obligatorisch und wird in anderen Publikationen weggelassen, so dass wir bekommen:

$$(364) \text{ att}(Q, K, V) = \text{softmax}(QK^T)V$$

Wir besprechen erstmal die Bedeutung dieser Funktion.

- QK^T liefert uns eine Matrix von gewichteten Summen der keys mit den queries.
- *softmax* ist hier wohl Zeilenweise zu verstehen, sprich: $\exp((QK^T)_{ij})$ wird dividiert durch $\sum_i \exp((QK^T)_{ij})$. In jedem Fall soll ein normalisierter Vektor multipliziert werden mit jedem einzelnen Value-Vektor in V .
- “Softmax creates a vector of similarity scores between the query q and the key vectors in K . The output of attention is thus a weighted sum of the value vectors where the weight for each value represents its relevance.
- Also: die Vektoren in $\text{softmax}(QK^T)$ sind die Gewichte der einzelnen Value-Vektoren

Diese Attention layers werden also übereinander gestapelt. Ich verstehe aber nicht, wo die Eingabe und wo die Ausgabe hineinkommt, die ja variabel sind. Die Größe der obigen Matrizen ist **fix**.

Mir bleibt aber die ganze Architektur

25 Transfer Learning – Fine-tuning

Bert, großes language model. Hieraus kann man viele kleine Aufgabe, wie POS-tagging, mit geringem Aufwand ableiten.

26 Metriken und Kostenfunktionen – extern

Metriken sind, im Gegensatz zu Kostenfunktionen, nicht für das Training, sondern zur Evaluation von Modellen gedacht. Wichtig: Accuracy, precision, recall, F-score.

Accuracy Akkuratheit liefert uns einfach (für Klassifikationsaufgaben). Also einfach folgende Formel:

$$(365) \quad acc(f, D) = \frac{\# \text{ falsche Klassifizierung von } f \text{ auf } D}{|D|}$$

Ganz interessant hierbei ist: wenn wir MSE applizieren auf eine 0,1 Klassifikation, dann bekommen wir genau dasselbe Ergebnis! Accuracy ist aber auf beliebig endlich vielen Klassen eindeutig definiert.

Frage: unter welchen Umständen ist accuracy wenig aussagekräftig?

Precision und Recall Nehmen wir an, wir haben eine Klassifikation nach $\{0, 1\}$. Wir definieren die Precision eines Klassifikators für Klasse 1 als

$$(366) \quad prec(f, D, 1) = \frac{\# \text{ richtige Klassifizierung von } 1 \text{ in } D}{\# \text{ Klassifizierung von } 1 \text{ in } D}$$

Precision sagt: wie genau ist unser Klassifikator für label 1, anders gesagt: wenn er 1 ausgibt, wie sicher können wir sein, dass es wirklich 1 ist? Nehmen wir an, wir haben sehr selten das label 1; unser Klassifikator wird immer 0 ausgeben. Er mag eine hohe accuracy haben, aber seine precision wird 0 sein!

Dual dazu wird der Recall definiert als:

$$(367) \quad rec(f, D, 1) = \frac{\# \text{ richtige Klassifizierung von } 1 \text{ in } D}{\# 1 \text{ in } D}$$

Recall sagt: wie sicher ist es, dass unserer Klassifizierer label 1 erkennt, wenn es auftritt? Anders gesagt: wie sicher können wir sein, dass er ein label 1 Datenpunkt richtig einordnet?

Diese Unterscheidung entspricht grob der Unterscheidung von **Typ I** und **Typ II** Fehlern in der Statistik; wir unterscheiden *falsche Negative* von

falschen Positiven. Was uns wichtiger ist, hängt vom Kontext ab. Wichtig: beide Maße sind asymmetrisch, setzen also eine besondere Kategorie voraus, die explizit gemacht werden muss. Damit ist das Maß auch implizit binär – es gibt nur eine Kategorie, die zählt, und die anderen.

F-score Der F-score (auch F1-score) mittelt nun über die beiden:

$$(368) \quad F(f, D, 1) = 2 \cdot \frac{rec(f, D, 1) \cdot prec(f, D, 1)}{rec(f, D, 1) + prec(f, D, 1)}$$

Da Precision und Recall in $[0, 1]$ liegen, folgt dass der Zähler in $[0, 1]$ liegt; der Nenner in $[0, 2]$. Um diese inhärente Asymmetrie auszugleichen wird der Term mit 2 multipliziert, also liegt der *F*-score ebenso in $[0, 1]$ (das kann man recht einfach zeigen).

Der F-score sorgt dafür, dass sowohl Precision als auch Recall berücksichtigt werden; wenn also ein Term 0 oder sehr niedrig ist, ist auch der F-score 0 bzw. sehr niedrig.

Confusion matrix Der F-score berücksichtigt gleichermaßen falsche Positive (Typ I Fehler) wie auch falsche Negative (Typ II Fehler). Allerdings erlaubt er, als einfache Kennzahl, keine Rekonstruktion der Zahlen, aus denen er sich errechnet. Die Art der Fehler kann man mit der confusion matrix oder **Wahrheitsmatrix** darstellen. Für einen binären Klassifikator, der nach $\{0, 1\}$ klassifiziert, kann man folgende Tabelle erstellen:

	Nach 0 klassifiziert	Nach 1 klassifiziert
Label 0	korrekte Negative (\approx precision)	# falsche Positive
Label 1	# falsche Negative	korrekte Positive (\approx recall)

Diese Tabelle lässt sich auch erweitern für eine beliebige Anzahl von Kategorien nach denen klassifiziert wird; so lässt sich aus dieser Tabelle ablesen, welche Kategorien auf welche Art falsch kategorisiert werden.

27 *No free lunch* – Gibt nix umsonst

27.1 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 Theoreme 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 das: 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.

27.2 Die NFL Theoreme und was sie bedeuten

Das Ziel (die objektive Funktion) Wir suchen eine Funktion

$$f : X \rightarrow Y$$

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

$$f(x) : Y \rightarrow [0, 1]$$

Natürlich ist leicht zu sehen, dass das nur eine Verallgemeinerung darstellt. Uns interessieren also Funktionen

$$f : (X \times Y) \rightarrow [0, 1]$$

so dass

$$\sum_{y \in Y} f(x, y) = 1, \text{ f.a. } x \in X$$

Wir bezeichnen die Menge dieser Verteilungen mit

$$\text{prob}(Y^X)$$

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 $f \in \text{prob}(X^Y)$ a priori 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

$$k : Y \times Y \rightarrow \mathbb{R}.$$

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

$$(369) \quad \delta(x, y) = \begin{cases} 1, & \text{falls } x = y \\ 0 & \text{andernfalls} \end{cases}$$

Weiterhin bezeichnen wir

- mit c eine beliebige Konstante;
- mit y_f das “objektiv richtige” y so dass (x, y_f) in unseren Daten D ;
- mit y_h die Vorhersage unseres Algorithmus.

Nimm nun die Funktion

$$(370) \quad \delta(c, k(y_h, y_f)) : Y^3 \rightarrow \{0, 1\}$$

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”:

$$(371) \quad \Lambda(c, y_h) = \sum_{y_f \in Y} \delta(c, k(y_h, y_f)) : Y^2 \rightarrow \{0, 1\}$$

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

$$(372) \quad \Lambda(c, y) = \Lambda(c, y')$$

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

$$(373) \quad \Lambda(c)$$

Eine einfache Funktion, die das nicht erfüllt, wäre z.B. eine Funktion

$$(374) \quad k(y_h, y_f) = \max(\epsilon, y_h) \cdot (y_h - y_f)^2, \quad \epsilon > 0$$

(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 Wahrscheinlichkeitsverteilung

$$\theta : X \rightarrow [0, 1],$$

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

$$(375) \quad P(D|f) = \prod_{(x,y) \in D} \theta(x) f(x, y)$$

(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 weiterhin brauchen ist der Algorithmus a . Der Algorithmus ändert in jedem Schritt seine vorher-sagen: initialisiert ist er eine Funktion

$$a_0 \in Y^X.$$

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:

$$(376) \quad k_1 := k(y_1, a_0(x_1))$$

dann ändert er sich zu $a_1 \in X^Y$ usw. Der Algorithmus ist erstmal eine Folge von Funktionen

$$\langle a_0 : X \rightarrow Y, a_1 : X \rightarrow Y, a_2 : X \rightarrow Y, \dots \rangle$$

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

$$(377) \quad k_2 = k(a_1(x_2), f(x_2))$$

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.

$$(378) \quad P(k = 0 | f, a, D)$$

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:

$$(379) \quad P(k = z | f, a, D) = \sum_{k(y, a_D(x))=z} \theta(x) f(x, y)$$

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?)

27.3 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, denn auf diesen Punkten können wir sehr leicht bessere/schlechtere Algorithmen ausmachen.

Das erste Wolpertsche NFL Theorem besagt (grob):

Theorem 13 (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 a_1, a_2 gilt:*

$$\sum_{f \in \text{prob}(Y^X)} P(k = z|f, D, a_1) = \sum_{f \in \text{prob}(Y^X)} P(k = z|f, D, a_2)$$

Das bedeutet: die Wahrscheinlichkeit, dass wir mit einem Algorithmus, gegebenem Trainingsset in Unkenntnis der objektiven Funktion als nächstes einen Fehler von k haben (z.B. $k = 0$, also richtig zu liegen), ist für *jeden Algorithmus 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 f s unterschiedlich beitragen, da sie den Daten unterschiedliche Wahrscheinlichkeiten zuweisen. Wir haben:

$$(380) \quad P(k = z|a, f, |D| = m) = \sum_{|D|=m} P(k = z|f, a, D)P(D|f)$$

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 14 (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:

$$\sum_{f \in \text{prob}(Y^X)} P(k = z|f, |D| = m, a1) = \sum_{f \in \text{prob}(Y^X)} P(k = z|f, |D| = m, a2)$$

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 $f \in Y^X$ konsistent nur geringere Kosten erzeugt als $a2$ (also konsistent besser ist), dann bedeutet das dass wir – vereinfacht gesprochen – eine entsprechende Funktion $f' \in \text{prob}(Y^X)$ haben, auf der $a1$ *konsistent schlechter* abschneidet.

Permutierbare Funktionen – ein notwendiges& hinreichendes Kriterium Man kann dieses Theorem noch etwas stärker formulieren: die Bedingung $f \in Y^X$ (also der gesamte Funktionenraum ist involviert) ist etwas zu schwach. Eine **Permutation** ist eine Funktion

$$\pi : M \rightarrow M, \text{ wobei } \pi^{-1} \circ \pi(x) = x$$

Also eine Bijektion auf einer Menge. Sei $F \subseteq Y^X$ eine Menge von Funktionen. Wir sagen diese Menge ist *geschlossen unter Permutation*, wenn folgende Bedingung erfüllt wird:

$$f \in F \implies f \circ \pi \in F$$

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

- $F_{mon} \subseteq Y^X$, 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 $n \notin \mathbb{N}$).

- Die Klasse der linearen Funktionen (oder Polynome) ist nicht geschlossen unter Permutation (nimm obiges Beispiel, das macht die Funktion unstetig)
- Die Klasse der Wahrscheinlichkeitsfunktion $f : X \rightarrow [0, 1]$ 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 *a priori*-Aussage über Eingaben – Ausgabe Relationen machen können. Nun lässt sich das Theorem stärker formulieren:

Theorem 15 (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 einen 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.

27.4 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 *a priori* 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.

27.5 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

$$f : M \rightarrow \{0, 1\}$$

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.

28 Übersicht: eine Art “pipeline”

Hier ein kleines Übersichtskapitel, denn die Methoden des Deep Learning kann man (wie immer) auf verschiedene Arbeitsschritte aufteilen. Wichtig ist: die Methoden selber unterscheiden sich nicht unbedingt, sondern das Ziel!

Ebenso: es ist nicht gesagt dass man immer alle Arbeitsschritte, wie hier beschrieben machen muss! Es ist nur mein Versuch, etwas Ordnung in die Reihenfolge der Arbeitsschritte zu bringen.

Pre-processing Schritte 1: das Übliche Wie immer im machine learning muss man die Daten aufräumen:

- Datenpunkte mit fehlenden Merkmalen rauswerfen oder alternativ das Merkmal rauswerfen, oder das Merkmal mit Median auffüllen etc.
- In linguistischen Anwendungen: fast immer tokenisieren, evtl. lemmatisieren, evtl. Satzzeichen entfernen etc.

Hier geht es darum, saubere Daten zu bekommen!

Pre-processing Schritte 2: embedding Hier sind v.a. *embeddings* zu sehen, im NLP also word-embeddings. Diese Techniken laufen durchaus unter deep learning, aber: dies ist ein Arbeitsschritt, der *vor* dem eigentlichen Training des Modells abgeschlossen sein muss. Es spricht auch nichts dagegen (im Gegenteil), ein bereits existierendes word-embedding für einen spezielleren task zu nutzen (wir POS-tagging): gute word embeddings sind ziemlich aufwändig!

Auch die Zieldaten müssen evtl. eingebettet werden; z.B. wenn wir 6 Klassen haben, machen wir daraus 6 *one-hot* Vektoren

Hier geht es darum, Daten zu bekommen, auf denen neuronale Netze operieren können

Pre-processing Schritte 3: Daten aufteilen Wichtig: im Deep Learning praktisch immer: Trainset, Devset, Testset. Üblicherweise 60-20-20

Das eigentliche Training Jetzt muss man ein passendes Modell wählen für die Aufgabe, und kann das Modell auf den Daten trainieren (Trainset)

Parallel oder unmittelbar danach wird evaluiert auf dem *Devset*.

Hyperparameter Tuning Nun werden verschiedene Hyperparameter getestet für die Performanz auf dem Devset:

- Architektur (CNN,RNN,LSTM,...)
- Anzahl der hidden layer
- Breite der layer
- Dropout
- ...

Am Ende gibt es hoffentlich einen mehr oder weniger klaren Sieger.

Evaluation *Erst jetzt* (und nicht früher) wird das Testset genommen, und nur der Sieger des Tuning wird darauf evaluiert. Hier gilt es natürlich, die passende Metrik zu finden. Vorsicht bei accuracy: kleine Zielklassen können verschwinden, also lieber noch F-score!

Bei accuracy wird verglichen mit *random baseline* (zufälliger Klassifikator) und *majority baseline* (klassifiziert immer zur größten Klasse), um die Nummer aussagekräftig zu machen.

Transfer Der letzte Schrei ist *transfer-learning*: es wird ein sehr komplexes und mächtiges Modell genommen wie **BERT**, und dann geschaut inwieweit einzelne layer von BERT gewissen Teilaufgaben des Gesamttasks übernehmen. Man kann als fertige komplexe Modelle nehmen, daraus einzelne Teile entnehmen und wiederum leicht nachtrainieren. So bekommt man sehr schnell und sehr einfach gute Ergebnisse. Das ist aber ein sehr junger Ansatz, siehe z.B. [1]!

29 Kleines Wörterbuch der linearen Algebra (und etwas mehr)

Körper sind kompliziert zu definieren (9 Axiome). Wir haben:

K1 Eine Menge M , abgeschlossen unter zwei Operationen $+$, \cdot

K2,3 Es gibt für $+$, \cdot jeweils ein neutrales Element, genannt $0, 1$

K4,5 Es gibt für $+$, \cdot und alle $x \in M - \{0\}$ jeweils ein inverses Element, genannt jeweils $-x, 1/x$, so dass $x + (-x) = 0, x \cdot (1/x) = 1$

K5-9 $\cdot, +$ sind kommutativ und assoziativ

K10 $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$ f.a. $a, b, c \in M$

Es reicht zu wissen dass es drei bekannte Körper gibt,

1. die rationalen,
2. die reellen und
3. die komplexen Zahlen

sowie Körper, die mittels algebraischen Konstruktion (z.B. Restklassen) aus diesen konstruiert werden.

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

$$\lambda \cdot (v_1, \dots, v_n), \text{ wobei } \lambda \in \mathbb{R},$$

mit

$$(381) \quad \lambda \cdot (v_1, \dots, v_n) = (\lambda v_1, \dots, \lambda v_n)$$

Die Addition zweier Vektoren ist definiert durch:

$$(382) \quad (v_1, \dots, v_n) + (w_1, \dots, w_n) = (v_1 + w_1, \dots, v_n + w_n)$$

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 $f : V \rightarrow V$ so dass

$$(383) \quad f(\vec{v}) = (\lambda \cdot \vec{v}) + \vec{w}$$

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.

Linear unabhängig Ein Vektor \vec{v} ist **linear abhängig** von einer Menge $\{\vec{w}_1, \dots, \vec{w}_i\}$, falls es $\lambda_1, \dots, \lambda_i \in \mathbb{R}$ gibt so dass

$$(384) \quad \vec{v} = \lambda_1 \vec{w}_1 + \dots \lambda_i \vec{w}_i$$

Man nennt das eine **Linearkombination** von $\vec{w}_1, \dots, \vec{w}_i$. Eine Menge $\{\vec{w}_1, \dots, \vec{w}_i\}$ ist **linear unabhängig**, falls \vec{w}_1 nicht linear abhängig ist von $\{\vec{w}_2, \dots, \vec{w}_i\}$ (die Auswahl von \vec{w}_1 ist beliebig und spielt keine Rolle). Was man wissen muss:

- Eine Menge von n linear unabhängigen Vektoren der Dimension m spannt einen n -dimensionalen Teilraum des Vektorraums $V \subseteq \mathbb{R}^m$ auf.
- Es gibt jeweils nur maximal m linear unabhängige Vektoren der Dimension m . Falls also bei 1 gilt: $n > m$, dann können die Vektoren gar nicht linear unabhängig sein!

Hyperebenen stellen eine Generalisierung von Ebenen (im 3-dimensionalen Raum) auf beliebige Dimensionen dar. Eine normale Ebene ist spezifiziert durch einen **Stützvektor** \vec{s} und zwei linear unabhängige **Richtungsvektoren** \vec{r}_1, \vec{r}_2 . Wichtig ist: die Richtungsvektoren müssen

1. Ungleich 0_V sein (sonst haben sie keine Richtung!)
2. Sie müssen linear unabhängig sein – sonst ist die Ebene unterdeterminiert!

Ein Punkt p liegt auf der Ebene, falls er sich darstellen lässt als

$$(385) \quad p = \lambda_1 \vec{r}_1 + \lambda_2 \vec{r}_2 + \vec{s} \text{ für } \lambda_1, \lambda_2 \in \mathbb{R}$$

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 $\vec{r}_1, \dots, \vec{r}_{n-1}$ (ungleich 0). Technisch gesehen ist die Hyperebene eine Menge von Punkten:

$$(386) \quad H = \{ \vec{s} + \lambda_1 \vec{r}_1 + \dots + \lambda_{n-1} \vec{r}_{n-1} : \lambda_1, \dots, \lambda_{n-1} \in \mathbb{R} \}$$

Norm : definiert auf einem Vektorraum V ist das eine Funktion

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

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:

$$\|(v_1, \dots, v_n)\| = \sqrt{v_1^2 + \dots + v_n^2}$$

In dieser geometrischen Interpretation wird Bedingung 3 zur **Dreiecksungleichung**: in jedem rechteckigen Dreieck ist die Länge der Hypotenuse 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:

$$\|(v_1, \dots, v_n)\|_p = (\sum_{i=1}^n |v_i|^p)^{\frac{1}{p}}$$

Für $p = 1$ vereinfacht sich das zu

$$(387) \quad \|(v_1, \dots, v_n)\|_1 = \sum_{i=1}^n |v_i|$$

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.

Welchen Unterschied machen unterschiedliche Normen? Wir betrachten das in der Ebene, weil hier die Dinge am Einfachsten sein. Man nehme den Einheitskreis um den Ursprung. Mit dem Satz des Pythagoras lässt sich leicht ableiten: sowohl $(0, 1)$ als auch $(\sqrt{0.5}, \sqrt{0.5})$ sind Vektoren, die vom Ursprung auf den Einheitskreis zeigen. Sie sind also gleich lang, rein geometrisch. (beachte: $0.5 < \sqrt{0.5} < 1!$) Es gilt auch ganz offensichtlich auch:

$$(388) \quad \|(\sqrt{0.5}, \sqrt{0.5})\|_2 = \|(0, 1)\|_2 = 1$$

Die euklidische Norm in der Ebene ist also ein Modell der geometrischen Länge. Dagegen gilt:

$$\begin{aligned} \|(\sqrt{0.5}, \sqrt{0.5})\|_3 &= (\sqrt{0.5}^3 + \sqrt{0.5}^3)^{1/3} \\ &= (0.5 \cdot \sqrt{0.5} + 0.5\sqrt{0.5})^{1/3} \\ &= ((\sqrt{0.5} + \sqrt{0.5}) \cdot 0.5)^{1/3} \\ &= ((2 \cdot \sqrt{0.5}) \cdot 0.5)^{1/3} \\ &= \sqrt{0.5}^{1/3} \\ &< 1 \end{aligned}$$

Dagegen:

$$\begin{aligned} \|(0, 1)\|_3 &= (0 + 1)^{1/3} \\ &= 1 \end{aligned}$$

Was kann man also sagen? Normen mit Grad $p > 2$ bestrafen einzelne große Komponenten stärker als gleich verteilte Komponenten! Kurz: Ausreißer werden bestraft!

Die Manhattan-Norm macht übrigens genau das Gegenteil:

$$(389) \quad \|(\sqrt{0.5}, \sqrt{0.5})\|_1 = \sqrt{0.5} + \sqrt{0.5} > 1 = \|(0, 1)\|_1$$

Hier belohnen wir ungleichmäßige Abweichungen (d.h. in möglichst wenigen Komponenten). Das kann durchaus Sinn machen, wenn eine einzelne Komponente gut vernachlässigt werden kann!

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

$$(390) \quad d(\vec{v}, \vec{w}) = \|\vec{v} - \vec{w}\|$$

wobei natürlich

$$(391) \quad \vec{v} - \vec{w} = (v_1 - w_1, \dots, v_i - w_i)$$

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

$$(392) \quad d_2(\vec{v}, \vec{w}) = \|\vec{v} - \vec{w}\|_2$$

Für die verschiedenen Distanzen gilt dann natürlich genau dasselbe wie für die verschiedenen Normen: je höher das p in d_p , desto stärker wird die Abweichung in einzelner Komponenten (gegenüber verteilter Abweichung in vielen Komponenten) bestraft. Neutral ist dabei die 2!

Kosinus (zweier Vektoren) Im rechtwinkligen Dreieck ist der Kosinus eines Winkels $\cos(\alpha)$ definiert als

$$(\text{Länge der Ankathete})/(\text{Länge der Hypotenuse})$$

also $\cos(90^\circ) = 0$, $\cos(0^\circ) = 1$, und alles andere liegt dazwischen. Man kann den Kosinus auch verallgemeinert definieren als Funktion $\cos : \mathbb{R} \rightarrow \mathbb{R}$, und zwar einigermaßen kompliziert als unendliche Reihe:

$$(393) \quad \cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{2n!}$$

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 Hypotenuse auffassen, von seinem Ende eine Linie ziehen so dass sie den (verlängerten) anderen Vektor im rechten Winkel schneidet, und dann die geometrische Definition anwenden.

Allgemeiner (d.h. in beliebigen Dimensionen) lässt sich der Cosinus wie folgt berechnen: wir haben

$$(394) \quad \vec{x}^\top \vec{y} = \|\vec{x}\|_2 \|\vec{y}\|_2 \cos \sphericalangle(\vec{x}, \vec{y})$$

Daraus folgt mittels Termumformung:

$$(395) \quad \cos \sphericalangle(\vec{x}, \vec{y}) = \frac{\vec{x}^\top \vec{y}}{\|\vec{x}\|_2 \|\vec{y}\|_2}$$

Skalarprodukt : Bislang haben wir Multiplikation nur mit Skalaren ausgeführt; es gab noch keine Möglichkeit, zwei Vektoren miteinander zu multiplizieren. Das macht das Skalarprodukt $\bullet : V^2 \rightarrow \mathbb{R}$. 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:

$$(396) \quad (a_1, \dots, a_n)^\top (b_1, \dots, b_n) = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

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 $\odot : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$, einfach definiert durch **punktweise Multiplikation**: $(x_1, \dots, x_n) \odot (y_1, \dots, y_n) = (x_1 y_1, \dots, x_n y_n)$. 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 $n = 1$. Matrizen schreibt man wie folgt:

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{pmatrix}$$

NB: 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 gewöhnungsbedürftig; wir haben normalerweise die Addition und die Multiplikation. Addition ist im Prinzip dasselbe wie bei Vektoren. Nimm zwei Matrizen $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$, dann haben wir

$$\mathbf{A} + \mathbf{B} \subseteq \mathbb{R}^{m \times n}, \text{ und } (\mathbf{A} + \mathbf{B})_{ij} = \mathbf{A}_{ij} + \mathbf{B}_{ij}.$$

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:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot (b_1 \quad b_2 \quad b_3) = \begin{pmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & a_1 \cdot b_3 \\ a_2 \cdot b_1 & a_2 \cdot b_2 & a_2 \cdot b_3 \\ a_3 \cdot b_1 & a_3 \cdot b_2 & a_3 \cdot b_3 \end{pmatrix}$$

Wenn wir das umdrehen, dann bekommen wir:

$$(a_1 \ a_2 \ a_3) \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = ((a_1 \cdot b_1) + (a_2 \cdot b_2) + (a_3 \cdot b_3))$$

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:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

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

$$\mathbf{A} \cdot \mathbf{1}_n = \mathbf{1}_m \cdot \mathbf{A} = \mathbf{A}$$

Eigenvektor und Eigenwert Der **Eigenvektor einer Matrix** M ist ein Vektor \vec{v} so dass gilt: $M\vec{v} = \lambda\vec{v}$ für einen Skalar λ . Also macht M mit \vec{v} nichts anderes als den Vektor skalieren, anstatt seine Richtung zu ändern. λ nennt man dann den **Eigenwert**. Es ist ein schwieriges Problem, die Eigenvektoren und -werte einer großen Matrix auszurechnen, dass man im allgemeinen Fall nur näherungsweise lösen kann.

Gradienten Gradienten sind eine Generalisierung von Ableitungen für multivariate Funktionen. Ableitungen für Funktionen

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

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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ eine Funktion, $i \in \{1, \dots, n\}$; dann haben wir mit

$$(397) \quad \frac{df}{dx_i} f(x_1, \dots, x_n)$$

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 wie 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ist eine Funktion $\nabla f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, dessen Ausgabekomponenten 1-n durch die partiellen Ableitungen von f nach x_1, \dots, x_n berechnet werden, d.h.

$$(398) \quad \nabla f(x_1, \dots, x_n) = \left(\frac{df}{dx_1} f(x_1, \dots, x_n), \dots, \frac{df}{dx_n} f(x_1, \dots, x_n) \right)$$

Alternativ kann man auch die Einheitsvektoren zur Definition nutzen; nenne die n -dimensionalen Einheitsvektoren mit i ter Komponente 1 $1_1, \dots, 1_n$; dann ist

$$(399) \quad \nabla f(x_1, \dots, x_n) = \frac{df}{dx_1} f(x_1, \dots, x_n) 1_1 + \dots + \frac{df}{dx_n} f(x_1, \dots, x_n) 1_n$$

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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$: 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 $\epsilon > 0$,

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

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.

(Einfaches) Perzeptron Ein einfaches Perzeptron hat die Form

$$(400) \quad P(\vec{x}) = g(M\vec{x} + \vec{a})$$

wobei $M\vec{x} + \vec{a}$ ein lineares Modell, g eine nichtlineare Funktion. g kann im erweiterten Fall auch die Form haben $g_1 \circ g_2$, z.B. $\text{softmax} \circ \text{ReLU}$. Entscheidend ist dass es nur ein lineares layer gibt.

Multilayer Perzeptron (MLP), auch genannt feedforward network

Ein MLP ist eine Funktion

$$(401) \quad MLP = P_1 \circ \dots \circ P_n$$

also eine Verkettung von Perzeptronen. Man nennt MLP auch feedforward Network; sie sind die einfachsten nichttrivialen neuronalen Netze. Das Beispiel aus 401 hat $n - 1$ *hidden layer*; also hat ein einfaches Perzeptron kein hidden layer. Das offene layer in *MLP* wäre dasjenige von P_1 .

Wenn man von einem Eingabelayer spricht, dann meint man nur soz. dummy-Neuronen, in welche die Werte der Eingabe geschrieben werden; in der funktionalen Schreibweise fällt dieses layer ersatzlos weg.

References

- [1] Ian Tenney, Dipanjan Das, and Ellie Pavlick. BERT rediscovers the classical NLP pipeline. *CoRR*, abs/1905.05950, 2019.