Neuronale Netze und Tiefe Architekturen

Christian Wurm
cwurm@phil.hhu.de

May 4, 2023

Abstract

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

Contents

1 Maschinelles Lernen: Klassifikation und Regression **6**
1.1 Test und Trainigsdaten, 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 Entropie etc. **26**
4.1 Bedingte Entropie 29
4.2 Kullback-Leibler-Divergenz 30
4.3 Kreuzentropie 31

5 Exkurs: Conditional Random Fields **32**
5.1 Definition 32
5.2 Ein Beispiel 34
5.3 Parameter schätzen 34

6 Lineare Modelle **35**
6.1 Die einfache lineare Regression 35
6.2 Der komplexe lineare Fall 37
6.3 Lineare Regression in R 38
6.4 Beschränkungen linearer Modelle, lineare Separierbarkeit . . . 40
6.5 Abschluss unter Komposition 42
6.6 Höhere Dimensionalität 44

7 Die Logistische Regression und das einfache Perzeptron **47**
7.1 Das einfache Perzeptron 47
7.2 Optimierung von einfachen Perzeptronen 49
7.3 Bedeutung 50
7.4 Höherdimensionale Modelle mit nichtlinearen Funktionen . . . 53
7.5 Softmax-Regression 54

8 Nichtlineare Funktionen: Aktivierung u.a. **58**
8.1 Überblick 58
8.2 Innere Funktionen (Aktivierungsfunktionen) 60
8.3 Punktweise Erweiterung von Funktionen 61
8.4 Rectified linear units, Max-out units 62
8.5 Ausgabefunktionen 64

9 Was können Tiefe Architekturen? **66**
9.1 Mächtigkeit ist nicht Lernbarkeit 66
9.2 Universelle Approximation 68
9.3 Boolesche Funktionen: Ein Beispiel 71

10 Das fundamentale Theorem von Tiefe und Breite **73**

11 Optimierung von Neuronalen Netzen **75**
11.1 Parameter und Freiheitsgrade 75
11.2 Vorspiel 1: Analytische Optimierung 77
11.3 Vorspiel 2: Konvexe Optimierung 78

12 Gradienten berechnen **80**

13 Gradientenbasierte Optimierung **81**
13.1 Gradienten und klassische Optimierung 81

14 Differenzieren – einige Faustregeln **82**

15 Tiefe Netze trainieren **85**
15.1 Parameter und Hyperparameter 85
15.2 Backpropagation 89
15.3 Die Fehlerfunktion 90
15.4 Backpropagation Übersicht 91
15.5 Berechnungsgraphen 92
15.6 Backpropagation – Verwaltung 93
15.7 Berechnungsgraphen – Knoten und Funktionen 94
15.8 Backpropagation – Der Feedforward-Step 95
15.9 Backward-propagation 97
15.10Backpropagation: Ausgabe 99
15.11Beweis von Satz 13 100
15.12Gradient descent 103
15.13Alternative: Momentum 104
15.14Zusammenfassung 105

16 Initialisierung **106**

17 Regularisierung **107**

18 Word embeddings **108**
18.1 Einleitung 108
18.2 Vorspiel: tf-idf Kodierung 109
18.3 Skip-gram Methode 110
18.4 Hierarchical Softmax: Die Optimierung 115
18.5 word2vec und GloVe 117
18.6 Sog. “Additive Kompositionalität” 118

18.7 ELMo . . . 119
18.8 BERT . . . 121

19 Recurrent Neural Networks 122
19.1 Motivation . . . 122
19.2 RNN – zwei Auffassungen . . . 122
19.3 Definition . . . 123
19.4 Das *bag of words*-Modell CBOW . . . 126
19.5 SRNN . . . 127
19.6 RNNs stapeln . . . 128
19.7 Training von RNNs . . . 130
19.8 Vanishing Gradient . . . 132

20 Architekturen mit *gates* – LSTM 133
20.1 Vorspiel . . . 133
20.2 LSTM – Definition . . . 136
20.3 Gated recurrent units . . . 139

21 Zwei Anwendungsbeispiele für RNN 141
21.1 RNN für *sequence labelling* . . . 142
21.2 RNNs als Sprachmodelle . . . 146

22 Convolutional Neural Networks 150
22.1 Convolution – was ist das? . . . 150
22.2 Convolution – Beispiel 1 . . . 151
22.3 Convolution – Beispiel 2 . . . 155
22.4 Motivation . . . 160

23 Sequence-to-sequence Modelle 161
23.1 Einfache Modelle . . . 161
23.2 Bidirektionale Modelle . . . 162
23.3 Vector to sequence (VtS) . . . 163
23.4 Encoder – decoder (ED) . . . 165

24 Attention 166
24.1 Intuition: Alinierung und Wahrscheinlichkeit . . . 166
24.2 Definition - Die erste Architektur . . . 169
24.3 Training . . . 172
24.4 Attention is all you need – der Transformer . . . 173

25 Neural Language Models, Sprachgenerierung **175**
25.1 Sprachgenerierung 175

26 ELMo, revisited **176**

27 Bert: Transfer Learning **180**

28 Generative Adversarial Models **181**

29 Metriken und Kostenfunktionen – extern **182**

30 Übersicht: eine Art "pipeline" **184**

31 Kleines Wörterbuch der linearen Algebra (und etwas mehr) **186**

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*:

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

Im Deep Learning geht es normalerweise um Klassifikationsaufgaben: wir mchten für eine unendliche Menge von möglichen Eingaben eine endliche Menge von Ausgaben haben. Wichtig hierbei ist: Neuronale Netzen klassifizieren mit Mitteln der Regression. Jedes neuronale Netz ist erstmal eine Funktion $N : \mathbb{R}^n \to \mathbb{R}^m$, 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 $f : \Sigma^* \to T^*$ (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. $D = \{(2, 6.4, 3.92), (4.9, 3.57, 2.51), ...\}$. 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

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

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

(1) $f(2, 6.4) = 3.92$ (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 $f : M \to N$ approximieren wollen, dann haben wir

$$h : M \to N$$

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 **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

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

induzieren, z.B. um die Korrelation

$$\text{Tagestemparatur} \sim \text{Straftaten}$$

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

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

Da D eine Teilmenge des **Funktionsgraphen** von f ist, ist unsere Aufgabe ist wie folgt umrissen:

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

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

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

(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.

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

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

$$x^2 + ax + b \tag{4}$$

Dabei gibt es folgendes zu beachten:

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

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

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

$$f(x) = ax + b + c\sin(x) \tag{5}$$

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 Trainigsdaten, 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 permament verfolgen wird, und das man nur mit ausgefeilten Methoden in den Griff bekommt. Overfitting kann man mit Übergeneralisierung (!) 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 partionieren unsere Daten in Test- und Trainingssatz, wobei der erstere viel zu klein ausfällt. Das wird dann allerdings immer wieder iteriert (z.B. ist jeder Datenpunkt einmal Trainingsatz). Das liefert jedesmal ein (etwas) anderes Modell, aber am Ende können wir über die Ergebnisse mitteln und so sehen, ob das Modell insgesamt stimmt.

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 Verhaltnisses 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, dass 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 dass, 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 Trainingssatz spalten. Auf diese Weise schließen wir overfitting für die Parameter aus. Eine andere Sache, die jedoch passieren kann, ist dass wir auf diese Art und Weise overfitting für die Hyperparameter bekommen; denn diese werden immer wieder auf demselben Datensatz trainiert. Das ist insbesondere ein Problem für Gebiete, auf denen Daten nur schwer zu bekommen sind. 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 "daneben-liegt" von der richtigen Vorhersage (Kosten $\cong$ Falschheit).

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

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

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

Unsere Zielfunktion ist also eine Funktion

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

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 $K : \mathbf{Y} \times \mathbf{Y} \to \mathbb{R}$. Damit können wir unser Problem etwas formaler auffassen: wir suchen dasjenige θ, so dass

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

minimal wird, wir suchen also

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

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

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

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

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

Dann bekommen wir

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

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. $X = \{N_1, ..., N_i\}$ (im endlichen Fall), und es gilt:

1. $N_1 \cup ... \cup N_i = M$
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

$0 \cong$ Unmöglichkeit
$1 \cong$ Sicherheit

Nun haben wir, aus logischen Gründen folgendes:

(2) $P(A) \leq P(A \text{ oder } B)$ 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) $\quad P(A \text{ und } B) \leq P(A)$ 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) $\quad n + 0 = n \qquad n \cdot 0 = 0$

Ebenso haben wir:

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

sowie:

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

sowie:

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

Wir haben also folgende Korrespondenz:

Konjunktion $\cong \cdot$
Disjunktion $\cong +$

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

sowie

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

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* $P : \mathfrak{A} \to [0, 1]$ *eine Wahrscheinlichkeitsfunktion, so dass*

1. $P(\Omega) = 1$*;*
2. $P(\emptyset) = 0$*, und*
3. *falls* $A_1, A_2, ..., A_n$ *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 $\overline{A}$ (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

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

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 $P(\overline{A})$ aus $P(A)$ lautet wie folgt:

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

Das lässt sich sehr einfach ableiten: wir haben

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

also:

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

3.8 Die Summenregel

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

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

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:

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

3.9 Die Produktregel

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

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

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

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

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

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

Intuitiv ist das wie folgt zu verstehen: wenn A und B eintreffen, dann bedeutet dass das 1. A eintrifft und 2. unter dieser Voraussetzung B eintrifft (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) $P(A \cap B|X) = P(A|BX)P(B|X) = P(B|AX)P(A|X)$

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

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 einen Seitenannahmen, doch dazu später mehr.

4 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

$$H_P(\omega) := P(\omega) \cdot -log(P(\omega)) \tag{17}$$

Die Entropie eines einzelnen Ereignisses ist normalerweise weniger interessant als die Entropie einer ganzen Verteilung P (über einen diskreten Raum Ω, geschrieben $H(P)$:

$$H(P) := -\sum_{\omega \in \Omega} P(\omega) log(P(\omega)) \tag{18}$$

Es ist leicht zu sehen dass das einfach die Summe der Entropie der Ereignisse ist; wir haben nur das minus ausgeklammert. Als Faustregel lässt sich sagen: in einem Raum mit n Ergebnissen ist die Entropie *maximal*, wenn alle Ereignisse die gleiche Wahrscheinlichkeit $1/n$ haben; sie wird minimal (geht gegen 0), falls es ein Ereignis gibt dessen Wahrscheinlichkeit gegen 1 geht. Das deckt sich mit unseren Intuitionen: je größer die Entropie, desto weniger Sicherheit haben wir, wie das Ergebnis sein wird. Z.B.: nehmen wir das Beispiel eines fairen Würfels; wir können die Entropie des zugehörigen Wahrscheinlichkeitsraumes wie folgt ausrechnen:

```
> x = 0 : 5
> for(i in 1 : 6){
+x[i] < −1/6 ∗ log(1/6)}
> sum(x)
[1] − 2.584963
```

(Wir verzichten darauf, die Entropie ins positive zu wenden). Wenn wir hingegen annehmen, 5 Seiten haben die Wahrscheinlichkeiten 1/10 und die 6 hat eine Wahrscheinlichkeit 1/2, dann bekommen wir:

```
> x = 0 : 5
> for(i in 1 : 5){
+x[i] < −1/10 ∗ log(1/10)}
> x[6] = 1/2 ∗ log(1/2)
> sum(x)
[1] − 2.160964
```

Andersrum gesagt: je größer die Entropie (einer Wahrscheinlichkeitsverteilung für ein Zufallsexperiment), desto größer der Informationsgewinn, der darin besteht das Ergebnis zu erfahren. Wichtig ist: Entropie ist immer unabhängig von den einzelnen Ergebnissen, es spielt also keine Rolle ob die 1 oder die 6 eine erhöhte Wahrscheinlichkeit hat. Alles was zählt ist eben die Ungewissheit; wir können das mit einem weiteren Versuch nachrechnen:

```
> x = 0 : 5
> for(iin1 : 3){
+x[i] < −1/10 ∗ log(1/10)}
> x[4] = (1/20) ∗ log(1/20)
> x[5] = (3/20) ∗ log(3/20)
> x[6] = 1/2 ∗ log(1/2)
> sum(x)
[1] − 1.777507
```

Die Entropie ist also weiter gesunken, denn wir haben die Wahrscheinlichkeiten weiter ungleich aufgeteilt zwischen 2 Ergebnissen: während also die Entropie für 1,2,3,6 gleich geblieben ist, ist sie für 4,5 lokal gesunken, also ist sie auch global gesunken. Man kann auch umgekehrt sagen: da die uniforme Wahrscheinlichkeitsverteilung für uns den *Mangel* an relevanter Information bezeichnet, gibt es die Korrelation

$$\text{maximale Entropie} \approx \text{maximale Unwissenheit}$$

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:

$$H(X) := -\sum_{x \in X} P(X = x) log(P(X = x)) \tag{19}$$

4.1 Bedingte Entropie

Die bedingte Entropie von zwei Variablen (über demselben Wahrscheinlichkeitsraum) ist wie folgt definiert (hier bedeutet $y \in Y$ soviel wie: y ist ein Wert, den Y annehmen kann):

$$H(X|Y) = \sum_{y \in Y} P(Y = y) H(X|Y = y) \quad (20)$$

Wenn wir diese Definition auflösen, bekommen wir:

$$H(X|Y) = \sum_{x \in X, y \in Y} P(X^{-1}(x) \cap Y^{-1}(y)) log \left(\frac{P(X^{-1}(x) \cap Y^{-1}(y))}{P(Y^{-1}(y))} \right) \quad (21)$$

Die bedingte Entropie ist also ein Maß dafür, wie stark die Werte einer Zufallsvariable Y die Werte einer Zufallsvariable X festlegen. Wenn der Wert von X durch den Wert von Y – egal wie er ist – immer festgelegt ist, dann ist

$$H(X|Y) = 0 \quad (22)$$

insbesondere also:

$$H(X|X) = 0 \quad (23)$$

Umgekehrt, falls der Wert von Y keinerlei Einfluss hat auf die Wahrscheinlichkeitsverteilung des Wertes von X, dann haben wir

$$H(X|Y) = H(X) \quad (24)$$

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:

$$H(X|Y) = H(\langle X, Y \rangle) - H(Y) \quad (25)$$

wobei $\langle X, Y \rangle$ eine neue Variable ist, mit

$$P(\langle X, Y \rangle = (x, y)) = P(X = x, Y = y) = P(X^{-1}(x) \cap Y^{-1}(y)) \quad (26)$$

Wir nehmen also die Entropie der **Verbundverteilung**, und ziehen die Entropie von $H(Y)$ ab.

4.2 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:

$$D_{KL}(P\|Q) = \sum_{\omega\in\Omega} P(\omega) log \frac{P(\omega)}{Q(\omega)} \tag{27}$$

An dieser Definition kann man ablesen:

1. $D_{KL}(P\|Q) = 0$ gdw. für alle $\omega \in \Omega$ gilt: $P(\omega) = Q(\omega)$; denn $log(1) = 0$.
2. In allen anderen Fällen ist $D_{KL}(P\|Q) > 0$ (das ist nicht wirklich leicht zu sehen).
3. $D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$, d.h. wir haben ein asymmetrisches Maß.
4. Man kann es jedoch symmetrisch machen auf folgende Art und Weise:

$$D_2(P\|Q) = D_{KL}(P\|Q) + D_{KL}(Q\|P) = D_2(Q\|P)$$

Sie gibt uns also ein Maß dafür, wie weit Q von P **entfernt** ist. Dadurch unterscheidet sie sich konzeptuell von $H(X|Y)$, dass bestimmt wie stark X von Y *determiniert* wird.

Man bezeichnet $D_{KL}(P\|Q)$ auch als den **Informationsgewinn**, den man mit P gegenüber Q erzielt. Wenn wir z.B. das obige Kodierungsbeispiel fortführen, dann sagt uns $D_{KL}(P\|Q)$, wieviel Platz wir (im Durchschnitt) verschwenden, wenn wir eine Kodierung auf Q basieren, während die zugrundeliegende Wahrscheinlichkeitsverteilung P ist.

Dementsprechen nutzt man $D_{KL}(P\|Q)$ oft im Kontext, wo P die tatsächliche Verteilung ist, Q unser Modell, dass wir geschätzt haben.

4.3 Kreuzentropie

Die Definition der Kreuzentropie ist nun

(28) $HK(X, P, Q) = H(X) + D(P\|Q)$

Äquivalent dazu gibt es (im diskreten Wahrscheinlichkeitsraum, den wir hier betrachten werden).

(29) $HK(X, P, Q) = -\sum_{x \in \Omega} P(X = x) \cdot log(Q(X = x))$

Man beachte: im Anwendungsfall ist P ist "korrekte Verteilung", dargestellt als ein Vektor

(30) $\vec{y} = (0, ..., 0, 1, 0, ..., 0)$

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 29 sich reduziert zu

(31) $HK(\vec{y}, \vec{\hat{y}} = -log(\vec{\hat{y}}_i)$, wobei $\vec{y}_i = 1$

Denn alle anderen Terme reduzieren sich auf 0. Man kann das übrigens auch als reine Vektor-Multiplikation schreiben:

(32) $HK(\vec{y}, \vec{\hat{y}}) = -(\vec{y}^\top log(\vec{\hat{y}}))$

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!

5 Exkurs: Conditional Random Fields

5.1 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.

$$P(Y_v|x, Y_{v'} : v \sim v') \tag{33}$$

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

$$P(Y_v|x, Y_{v'} : v \sim v', x) = P(Y_v|x, Y_{v'} : v \neq v') \tag{34}$$

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

(35) $P(\vec{y}|\vec{x})$

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

$$(36) \quad score(Ys = y|x) = \sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j f_j(x, x_{j:i\sim j})$$

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

$$(37) \quad score(Y_i = y|s_{in}) = \sum_{j=1}^{m} \lambda_j f_j(s_{in}, y, Y_j = y_j : i \sim j)$$

Wir haben hier also das label der i-ten Stelle, dass 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:

$$(38) \quad score(s_{out}|s_{in}) = \sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j f_j(s_{in}, y, Y_j = y_j : i \sim j)$$

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

$$(39) \quad P(s_{out}|s_{in}) = \frac{exp(score(s_{out}|s_{in}))}{\sum_{s'_{out}} exp(score(s_{out}|s_{in}))}$$

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

$$(40) \quad score(Y_1 = y_1|x_1) = \sum_{j=1}^{m} \lambda_j f_j(y_1, x_i, y_i)$$

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!

5.2 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 ????

5.3 Parameter schätzen

6 Lineare Modelle

6.1 Die einfache lineare Regression

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

$$f(x) = ax + b \tag{41}$$

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

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

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

$$argmin_{a,b\in\mathbb{R}} \sum_{(x,y)\in D} (ax + b - y)^2 \tag{42}$$

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

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

$$\sum_{(a,b)\in D} (ax + y - b)^2 = (a_1x + y - b_1)^2 + ... + (a_nx + y - b_n)^2 \tag{43}$$

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

$$(a_1x + y - b_1)^2 = a_1^1x^2 + y^2 + a_1xy - a_1b_1x - b_1y + b_1^2 \tag{44}$$

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

$$\sum_{(a,b)\in D} (ax + y - b)^2 = ax^2 + by^2 + cxy + dx + ey + f \tag{45}$$

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

$$argmin_{x,y\in\mathbb{R}} ax^2 + by^2 + cxy + dx + ey + f \tag{46}$$

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

$$\nabla f(x, y) = ((2ax + cy + d), (2by + cx + e)) \tag{47}$$

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

$$2ax + cy + d = 0 \tag{48}$$

$$2by + cx + e = 0 \tag{49}$$

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

$$x = -\frac{cy + d}{2a} \tag{50}$$

also

$$2by - \frac{c^2y + cd}{2a} + e = 0 \leftrightarrow \tag{51}$$

$$y = \frac{cd - 2ae}{4ab - c^2} \tag{52}$$

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

6.2 Der komplexe lineare Fall

Im komplexeren lineare Fall nehmen wir an, dass

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0 \quad (53)$$

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

$$f(x_0, ..., x_n) = \sum_{(a,b) \in D} (a^n x_n + a^{n-1} x_{n-1} + ... + x_0 - b)^2 \quad (54)$$

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 $(a^n, a^{n-1}..., a)$

Wir müssen nun

$$\nabla f : \mathbb{R}^{n+1} \to \mathbb{R}^{n+1} \quad (55)$$

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

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

$$f(x_1, ..., x_n) = a_1 x_1 + a_2 x_2 + ... + a_n x_n + b \quad (56)$$

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

6.3 Lineare Regression in R

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

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

Denn hiermit kann man eine einfache Funktion nutzen:

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

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

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

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:

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

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

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

In unserem Fall also:

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

Man kann die Regressionsfunktion auch schön visualisieren, mittels:

> abline(lm1, col = "red")

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

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

Jetzt müssen wir nur noch den passenden Vektor z erstellen:

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

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

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

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

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

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

(60) $\phi(x, y) = (x \cdot y, x + y)$

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!

6.5 Abschluss unter Komposition

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

$$f(x) = ax + b \tag{61}$$

$$g(x) = cx + d \tag{62}$$

Nun ist

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

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

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

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

$$l_1(x) \tag{63}$$

$$n_1 \circ l_1(x) \tag{64}$$

$$l_2 \circ n_1 \circ l_1(x) \tag{65}$$

$$n_2 \circ l_2 \circ n_1 \circ l_1(x) \tag{66}$$

$$... \tag{67}$$

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.

6.6 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

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

aufgefasst. Wir werden jetzt die allgemeinere Perspektive nehmen mit

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

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

$$L(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{a} \tag{68}$$

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

$$\mathbf{M} = \begin{pmatrix} 2.3 & 1.2 & 4 \\ 1 & 3.7 & 10 \end{pmatrix} \tag{69}$$

$$\mathbf{a} = (5, 6.3) \tag{70}$$

Dann haben wir also:

$$L(\mathbf{x}) = (2.3x_1 + 1.2x_2 + 4x_3, 1x_1 + 3.7x_2 + 10x_3) + (5, 6.3) \tag{71}$$

$$= (2.3x_1 + 1.2x_2 + 4x_3 + 5, 1x_1 + 3.7x_2 + 10x_3 + 6.3) \tag{72}$$

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

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

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

$$(73) \quad argmin_{M,\vec{b}} \sum_{(\vec{v},\vec{w}) \in D} d(M\vec{v} + \vec{b})$$

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

$$(74) \quad argmin_{M,\vec{b}} \sum_{(\vec{v},\vec{w}) \in D} \|(M\vec{v} - \vec{b})\|_2$$

Wie kann man diese Funktion minimieren? Die überraschende Antwort lautet: die Regression einer Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ 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 $D_1 = \{(\vec{v}_1, (\vec{w}_1)_1), ..., (\vec{v}_i, (\vec{w}_i)_1)\}$
- ...
- Wir erhalten $M_{m,}$ und $\mathbf{b}_m$ (also erste Zeile von M, erste Komponente von $\vec{b}$) als eine einfache lineare Regression über $D_1 = \{(\vec{v}_1, (\vec{w}_1)_m), ..., (\vec{v}_i, (\vec{w}_i)_m)\}$

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 letztenendes, 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

   $$L_1 \circ L_2(\mathbf{x}) = L_1(L_2(\mathbf{x})) \tag{75}$$

2. Wenn wir Gleichung 71 betrachten, dann können wir mit elementaren Methoden die optimale Lösung finden. Das bedeutet: das Optimierungsproblem ist lösbar, wir finden immer die beste Lösung.

7 Die Logistische Regression und das einfache Perzeptron

7.1 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

$$S(x) = \frac{1}{1+e^{-x}} \tag{76}$$

Einige Beobachtungen: wir haben

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

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

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

$$T(x) = \frac{x}{1+|x|} \tag{77}$$

$$U(x) = \frac{x}{1+x^2} \tag{78}$$

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 - \frac{2}{e^{2x}+1} \tag{79}$$

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

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

$$P(X = A|x) = S(f(x)) \tag{80}$$

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

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

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

$$C(x) = \begin{cases} A, \text{ falls } P(X = A|x) > 0.5 \\ B \text{ andernfalls} \end{cases} \tag{81}$$

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

7.2 Optimierung von einfachen Perzeptronen

Das Prinzip des Perzeptrons ist einfach: aus der Wahrscheinlichkeit $P(A|\vec{d})$ wird eine Klasse A oder $\overline{A}$. Aber warum brauche wir überhaupt die Wahrscheinlichkeit? Die Antwort ist klar: einen diskreten Klassifikator kann man nicht optimieren; man braucht eine stetige, ableitbare Funktion, v.a. ohne Plateaus. Die Wahrscheinlichkeit nutzt man, um die Kostenfunktion zu definieren. Diese Methode ist ziemlich universell und wird auch z.B. im Deep Learning genutzt.

Log-loss Methode Nimm einen Prä-Klassifikator, der jeder Kategorie $A_i \in \mathbf{C}$, gegeben einen Vektor $\vec{d}$, eine Wahrscheinlichkeit $\hat{P}(A_i|\vec{d})$ zuweist. Der log-loss des Klassifikators auf Datenpunkt $(\vec{d}, A_j)$ (wobei also A_j die korrekte Kategorie für $\vec{d}$ ist) ist definiert als

$$-log\hat{P}(A_j|\vec{d}) \tag{82}$$

also der negative Logarithmus der zugewiesenen Wahrscheinlichkeit des korrekten labels.

Wenn man das in allgemeine Form bringen will, dann kann man den Datensatz D auffassen als eine (partielle) Wahrscheinlichkeitsfunktion

$$P_D(A_i|\vec{d}) \mapsto \{0,1\}, \text{ wobei } \vec{d}, A \in D \text{ für ein } A \tag{83}$$

Wir haben also eine spezielle Wahrscheinlichkeitsfunktion.

Binärer log-loss Die Kostenfunktion ist damit wie folgt definiert:

$$-\sum_{\vec{d}\in D} log(P_D(A_i|\vec{d}) \cdot \hat{P}(A|\vec{d}) + P_D(\overline{A}|\vec{d}) \cdot \hat{P}(\overline{A}|\vec{d})) \tag{84}$$

Das bedeutet: einer der beiden Terme ist immer null, der andere ist eben die log-Wahrscheinlichkeit die dem korrekten label zugewiesen wird.

7.3 Bedeutung

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

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

Wir tun das aber mittels zweier Zwischenschritte:

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

Die zugrundeliegende Annahme ist hierbei, dass es einen kritischen Bereich gibt, auf dem die Klassifikation von 0 zu 1 springt und umgekehrt ist; aber damit man das sieht, muss man zunächst eine **numerische Transformation** durchführen, nämlich die lineare Regression. 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

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

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 $\alpha_1, ..., \alpha_n$, 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, $f : \mathbb{R}^n \to \mathbb{R}$ 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:
>
> $C(\vec{x}) = 1$ gdw. $\vec{x}$ 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

(85) $H = \{\vec{s} + \lambda_1 \vec{r_1} + ... + \lambda_{n-1} \vec{r_n} : \lambda_1, ..., \lambda_n \in \mathbb{R}\}$

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

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

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

(87) $a_1 y_1 + ... + a_n y_n + a_{n+1} > 0$

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

7.4 Höherdimensionale Modelle mit nichtlinearen Funktionen

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

(88) $per(\vec{x}) = n(M\vec{x} + \vec{b}) = (n((M\vec{x} + \vec{b})_1, ..., n(M\vec{x} + \vec{b})_m)$

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

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

als eine unabhängige Applikation verschiedener einfacher linearer Funktionen. Das bedeudet: $per(\vec{x})$ 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 $per : \mathbb{R}^n \to \mathbb{R}^m$ 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.

7.5 Softmax-Regression

Die logistische Regression liefert uns eine Funktion $f : \mathbb{R}^n \to [0, 1]$; wir haben gesagt, wir können diese Funktion auffassen als

1. Wahrscheinlichkeit – mit der Beschränkung dass wir nur die Wahrscheinlichkeit eines *einzigen* Ereignisses A bekommen.

2. Klassifizierer – mit der Beschränkung, dass wir nur binär klassifizieren können: nämlich entweder A (entspricht $P(A|x) > 0.5$) oder andernfalls $\overline{A}$.

Wir bleiben mit der Frage zurück:

> *Wie kann man eine Regression zum klassifizieren über eine beliebige endliche Menge C nutzen?*

Die **Softmax-Funktion** ist eine sehr beliebte Funktion, mit der man eine n-dimensionale Eingabe in eine diskrete Wahrscheinlichkeitsverteilung über n Ergebnisse transformieren kann. Eine solche Funktion ist insbesondere wichtig wenn wir eine logistische Regression zur Klassifizierung einsetzen möchten. Nehmen wir einmal folgendes Beispiel.

Beispiel Wir möchten eine Eingabe $\vec{x} \in \mathbb{R}^n$ als eine Klasse $c \in C$ klassifizieren, wobei $|C|$ endlich ist. $\vec{x}$ kann ein Bild sein (Pixeldaten) und C eine Menge mit Objekten, oder $\vec{x}$ kann (wie im Boston housing) ein Vektor mit verschiedenen Daten sein und C eine Klassifizierung wie "gute Lange", "mittlere Lage" etc. Im Deep Learning wird man sehen dass $\vec{x}$ auch die Repräsentation eines Wortes sein kann, C eine Menge von POS-tags; im modernen machine learning werden praktische alle Eingaben erstmal zu Vektoren transformiert.

Wir kann man das mit logistischer Regression modellieren? Den Fall $|C| = 2$ haben wir bereits gesehen; aber der lässt sich hier nicht verallgemeinern. Was man stattdessen macht ist folgendes: man trainiert ein lineares Modell

$$f : \mathbb{R}^n \to \mathbb{R}^{|C|} \tag{90}$$

so dass Komponente der Ausgabe eine Kategorie C repräsentiert. Am konzeptuell einfachsten wäre es dann, einfach eine *argmax*-Funktion nachzuschalten:

$$max(x_1, ..., x_m) = max\{x_1, ..., x_m\} \tag{91}$$

Wenn wir also unsere Klassen nummerieren, dann bekommen wir,

$$C(\vec{x}) = c_i, \text{ wobei } i = argmax_i(\mathbf{A}(\vec{x}))_i \tag{92}$$

Anders gesagt: i ist die größte Komponente.

Das Problem ist folgendes: die *max*-Funktion ist nicht differenzierbar. Weiterhin gilt: selbst wenn wir an einem Punkt einen Gradienten bekommen, dann haben wir wahrscheinlich eine Ebene – ein minimale Änderung in der Eingabe verursacht keine Änderung in der Ausgabe. Solche Plateaus können es schwierig bis unmöglich machen, ein Modell zu trainieren. Die Werte ändern sich ja sprunghaft, also wie sollen wir numerische Parameter trainieren?

Daher gibt es die sog. *softmax*-Funktion, die eine ähnliche Funktion übernimmt, aber differenzierbar ist (daher *soft*, weil sie keine Knicke hat). Diese Funktion spielt auch im Deep Learning eine sehr wichtige Rolle, und eigentlich immer, wenn (nicht-binäre) Klassifizierung mit numerischen Berechnungen durchgeführt werden soll. Es gilt:

$$softmax : \mathbb{R}^n \to \mathbb{R}^n$$

wobei

$$softmax(x_1, ..., x_n)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{93}$$

Diese Notation ist gewählt um die Definition übersichtlicher zu machen; man kann auch äquivalent schreiben:

$$softmax(x_1, ..., x_n) = (\frac{e^{x_1}}{\sum_{j=1}^{n} e^{x_j}}, ..., \frac{e^{x_n}}{\sum_{j=1}^{n} e^{x_j}}) \tag{94}$$

Es gilt also, da wir durch die Summe der Komponenten dividieren:

$$\sum(softmax(\vec{x})) = 1 \tag{95}$$

Wir können den Vektor also als Wahrscheinlichkeitsfunktion lesen!

Wie gesagt: falls $\mathbf{x} \in \mathbb{R}^n$, dann ist auch $softmax(\mathbf{x}) \in \mathbb{R}^n$. Die softmax-Funktion ist eine Art "weiche" argmax-Funktion:

$\Rightarrow$ dadurch dass wir exponentieren bekommen alle Komponenten, die nicht unbedeutend kleiner sind als der Größte, eine vernachlässigbare Größe

Der große Vortei von softmax ist: es ist eine stetig differenzierbare Funktion. Das ist sehr wichtig für die Optimierung, für die wir Gradienten brauchen.

Optimierung von Softmax: Kreuzentropie, der allgemeine log-loss
Softmax-Funktionen werden üblicherweise als Ausgabefunktionen verwendet: sie liefern tatsächlich eine Art **konditionale Wahrscheinlichkeitsverteilung**:

$$P(c_i|\vec{x}) = softmax(\mathbf{A}(\vec{x}))_i \tag{96}$$

Am Ende nehmen wir natürlich wiederum die Komponente mit dem maximalen Wert. Wichtig ist dass die Funktion stetig und differenzierbar ist – andernfalls könnten wir nicht weiter optimieren, der Gradient gibt uns keinerlei Information.

Die Optimierung läuft (wie bei logistischer Regression) nicht auf die Ausgabe (diskret), sondern über die gesamte Wahrscheinlichkeitsverteilung. Hier nutzt man eine Kostenfunktion wie **categorial crossentropy** (Kreuzentropie), was in einer stetigen Kostenfunktion resultiert:

$$H(X, P, Q) = H(X_P) + D(P\|Q) = -\sum_{x\in\Omega} P(X = x) log Q(X = x) \tag{97}$$

wobei P die Verteilung unseres Systems ist, Q die "wahre Verteilung" (ergibt sich aus den "korrekten Vorhersagen". Das resultiert also in einer differenzierbaren Funktion die heuristisch (analytisch?) optimiert werden kann.

Wenn wir die obige Formel der Kreuzentropie auf eine Vorhersage und einen Datenpunkt applizieren, ist das viel einfacher als es aussieht:

- Sei $M : \mathbb{R}^n \to \mathbb{R}^i$ unser Machine Learning Modell (softmax regression).
- $M(\vec{x}) = \vec{y} \in \mathbb{R}^i$ ist ein Vektor, der eine Wahrscheinlichkeitsverteilung über i Klassen darstellt.
- Der "target"-Vektor bekommt dann die Form $\hat{y} \in \{0, 1\}^i$, wobei $|\hat{y}| = 1$ (*one-hot Vektor* sagt man auch dazu, entspricht einer Verteilung ohne Entropie).

Die Kreuzentropie der beiden Verteilungen reduziert sich dann auf

$$-log\ y^\top \hat{y} \tag{98}$$

Sei $\hat{y}_i = 1$. Wir bekommen dann (noch einfacher):

$$(99) \quad -log\ y^\top \hat{y} = -log(\vec{y}_i)$$

als unsere Kosten auf einem Datenpunkt, nämlich:

$\Longrightarrow$ den negativen Logarithmus der vorhergesagten Wahrscheinlichkeit des korrekten labels.

Wir optimieren also ausschliesslich, indem wir die Wahrscheinlichkeit unserer Vorhersage für das korrekte Ziellabel erhöhen – alles andere spielt keine Rolle!

Auf dem gesamten Datensatz betrachtet:

$$(100) \quad -\sum_{(\vec{x},\hat{y})\in D} log\ M(\vec{x})^\top \hat{y}$$

Das ist also die allgemeine Form der log-loss Funktion.

8 Nichtlineare Funktionen: Aktivierung u.a.

8.1 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. $S(x_1, ..., x_n) = (S(x_1), ..., S(x_n))$.

Gerade letzteres gilt für äußere Funktionen üblicherweise nicht!

Für hidden layer 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

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

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

$$A : \mathbb{R}^n \to \mathbb{R} \tag{101}$$

8.2 Innere Funktionen (Aktivierungsfunktionen)

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

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

$$S(x) = \frac{1}{1 + e^{-x}} \tag{102}$$

Einige Beobachtungen: wir haben

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

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

$$T(x) = \frac{x}{1 + |x|} \tag{103}$$

$$U(x) = \frac{x}{1 + x^2} \tag{104}$$

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 - \frac{2}{e^{2x} + 1} \tag{105}$$

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

8.3 Punktweise Erweiterung von Funktionen

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

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

Wir können das erweitern auf beliebige

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

mittels der Gleichung

(106) $f(x_1, ..., x_m) = (f(x_1), ..., f(x_m))$

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

8.4 Rectified linear units, Max-out units

ReLUs sind definiert durch:

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

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:

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

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:

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

   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 $G_1, ..., G_k$, wobei also

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

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

Die Ausgabe der maxout Funktion ist wiederum ein Vektor, wobei

(110) $g : \mathbb{R}^n \to \mathbb{R}^j$

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

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

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

8.5 Ausgabefunktionen

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

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

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

(113) $kompmax(x_1, ..., x_m) = i : x_i = max\{x_1, ..., x_m\}$

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

(114) $kompmax(N(\vec{y})) = C(\vec{y})$

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

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

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):

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

wobei

(117) $N(\vec{z}) = \vec{x}$

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

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

so dass (im Idealfall)

$$(119)\quad \sum_{y \in Y} P(y, \mathbf{x}) = 1$$

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).

$$(120)\quad softmax(x_1, ..., x_n)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

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 $softmax(\mathbf{x}) \in \mathbb{R}^n$. Wir ändern die Dimension also nicht!
- Wir haben $\sum(softmax(\vec{x})) = 1$, 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 Vortei von softmax ist: es ist eine kontinuierliche Funktion. Das ist sehr wichtig für die Optimierung, für die wir Gradienten brauchen. Softmax-Funktionen werden üblicherweise als Ausgabefunktionen verwendet: sie liefern tatsächlich eine Wahrscheinlichkeitsverteilung wie in (119). 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.

9 Was können Tiefe Architekturen?

9.1 Mächtigkeit ist nicht Lernbarkeit

Es gibt zwei Fragen, dir 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 um bedingt am besten geeignet, die Funktion zu verstehen und zu berechnen. Neuronale Netze versteht man am besten, wenn man sie als eine Abfolge von Schichten auffasst, wobei am Ende jede Schicht nacheinander appliziert wird (wir schauen uns das später im Detail an).

Sei **a** der Eingabevektor, M eine Matrix, **b** ein Vektor der passenden Länge, und g eine (nichtlineare) Aktivierungsfunktion. Ein Netz N der Tiefe 1 berechnet nun die Funktion

$$N(\vec{x}) = g(M\vec{x} + \vec{b}) \tag{121}$$

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

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

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 $N(\vec{x}) = N_i \circ ... \circ N_1(\vec{x})$ hat $i - 1$ *hidden layer*

9.2 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 $f : \mathbb{R} \to \mathbb{R}$. Das **Urbild** einer Menge $M \subseteq \mathbb{R}$ ist dann definiert als

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

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)*

$$f(x) = \begin{cases} 0, \textit{ falls } x \in \mathbb{Q} \\ 1 \textit{ andernfalls} \end{cases} \tag{124}$$

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

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

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).

9.3 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:

$$M_1(1,0) = (1,0,-1,0) \tag{125}$$
$$M_1(0,1) = (0,1,0,-1) \tag{126}$$
$$M_1(1,1) = (1,1,-1,-1) \tag{127}$$
$$M_1(0,0) = (0,0,0,0) \tag{128}$$

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:

$$g \circ M_1(1,0) = (1,0) \tag{129}$$
$$g \circ M_1(0,1) = (1,0) \tag{130}$$
$$g \circ M_1(1,1) = (1,-1) \tag{131}$$
$$g \circ M_1(0,0) = (0,0) \tag{132}$$

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

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

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

$$M_2 \circ g \circ M_1(1,0) = 1 \tag{133}$$
$$M_2 \circ g \circ M_1(0,1) = 1 \tag{134}$$
$$M_2 \circ g \circ M_1(1,1) = 0 \tag{135}$$
$$M_2 \circ g \circ M_1(0,0) = 0 \tag{136}$$

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 Geschicht 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!

10 Das fundamentale Theorem von Tiefe und Breite

Ein Lebesgue-Maß $\lambda : \mathbb{R}^n \to \mathbb{R}$ ist ein generalisiertes Maß von Volumen. Es hat die Eigenschaft, dass es n-dimensionalen Hyperrechtecken ihr n-dimensionales Volumen zuordnet.

Eine Menge von Punkten ist eine Lebesgue 0 Menge (in einem Raum), falls sie nach keiner Richtung eine Ausdehnung aufweist: für jede infinitesimale Positionsänderung gilt: wir verlassen die Menge. Das sind z.B.

- eine endliche Zahlenmenge in den reellen Zahlen
- aber auch die rationalen Zahlen in den reellen Zahlen
- jede Menge, die ein offenes/geschlossenes Intervall enthält, hat ein Maß größer als 0

Theorem 10 *(Fundamental Theorem of Network Capacity) Nimm zwei MLP mit identischen Parametern N, M, Y.*

1. *ein tiefes Netz mit $d = log_2 N$ layern, und Matrix-Dimensionen $r_{1,l}$, und*
2. *ein flaches Netz mit einem hidden layer und Matrix-Dimensionen $r_{2,l}$.*

Sei $r = min(r_{1,0}, M)$ und nimm an $r_{2,0} < r^{N/2}$.

Sei S die Menge der Parameter des tiefen Netzes, mit welchen es eine Funktion berechnet, die auch von dem flachen MLP berechnet werden kann. Dann hat S das Lebesgue-Maß 0.

Das besagt soviel wie: bei gleicher Breite nimmt die Berechnungskraft von tiefen Netzen unverhältnismäßig zu. Das betrifft aber einen Extremfall, nämlich 1 bzw. $log_2 N$ hidden layer. Hier die verallgemeinerte Version:

Theorem 11 *(Generalized Theorem of Network Capacity) Nimm zwei MLP mit identischen Parametern N, M, Y.*

1. *ein Netz mit Tiefe d_1 und Matrix-Dimensionen $r_{1,l}$, und*
2. *ein flacheres Netz mit Tiefe $d_2 < d_1$ und Matrix-Dimensionen $r_{2,l}$.*

Sei

$$r = min\{M, r_{1,0}, ..., r_{1,d_2 1}\}$$

und sei

$$r_{2,d_2-1} < r^{N/2^{d_2}}$$

Sei S die Menge der Parameter des tiefen Netzes, mit welchen es eine Funktion berechnet, die auch von dem flachen MLP berechnet werden kann. Dann hat S das Lebesgue-Maß 0.

11 Optimierung von Neuronalen Netzen

11.1 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 nenn daher den Parameter m auch den **Freiheitsgrad** des Models. Wir haben also ein Model

$$(\vec{x} \in \mathbb{R}^n) \overset{(M_1 \in \mathbb{R}^{m_1 \times n})}{\longrightarrow} (\vec{y_1} \in \mathbb{R}^{m_1}) \overset{g_1}{\longrightarrow} (\vec{y_2} \in \mathbb{R}^{m_2}) ....$$

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

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

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

(137) $4 \cdot 5 + 5 \cdot 4 = 40$

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

(138) $4 \cdot 10 + 10 \cdot 4 = 80$

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

(139) $4 \cdot 5 + 5 \cdot 5 + 5 \cdot 4 = 65$

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 – bei gleichbleibender Anzahl von Neuronen!

11.2 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.

11.3 Vorspiel 2: Konvexe Optimierung

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

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

Allgemeiner:

Definition 12 *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:

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

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**

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

ist konvex, falls

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

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

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

Das bedeutet natürlich einen enormen Vorteil: um

(141) $argmin f(x)$

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

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

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

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

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

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

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

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

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

und wir suchen

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

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

12 Gradienten berechnen

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

$$\frac{df}{dx_i} f(x_1, ..., x_i, ..., x_n) = \frac{df}{dx} f(a_1, ..., x, ..., a_n) \tag{145}$$

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

$$\frac{df}{dx} x^2 + 2y = 2x \tag{146}$$

und

$$\frac{df}{dy} x^2 + 2y = 2 \tag{147}$$

Allerdings gilt:

$$\frac{df}{dx} x^2 \cdot 2y = 2x \cdot 2y \tag{148}$$

und

$$\frac{df}{dy} x^2 \cdot 2y = 2x^2 \tag{149}$$

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.

13 Gradientenbasierte Optimierung

13.1 Gradienten und klassische Optimierung

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

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

gegeben dass

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

(152) $$...$$

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

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

Wir fangen an einem (relativ) beliebigen Punkt

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

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

(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.

14 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**:

$$(156)\quad (f(x) + (g(x))' = f(x)' + g(x)'$$

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

$$(157)\quad \frac{\partial f}{\partial x}(f(x,\vec{y}) + g(x,\vec{y})) = \frac{\partial f}{\partial x}f(x,\vec{y}) + \frac{\partial f}{\partial x}g(x,\vec{y})$$

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

$$(158)\quad \nabla(f(\vec{x}) + g(\vec{x})) = \nabla f(\vec{x}) + \nabla g(\vec{x})$$

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

$$(159)\quad f \circ g(x) = f(g(x))$$

Die Regel lautet:

$$(160)\quad (f \circ g(x))' = f'(g(x)) \cdot g'(x)$$

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

$$(161)\quad (\frac{1}{x})' = -\frac{1}{x^2}$$

Wenn wir nun die Ableitung von

$$(162)\quad f(x) = \frac{1}{3x^2}$$

berechnen wollen, geht das mittels

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

Also haben wir

$$f'(x) = -\frac{1}{(3x^2)^2} \cdot 6x = -\frac{6x}{9x^4} = -\frac{2}{3x^3} \tag{164}$$

Wo kommt diese Regel her? Erinnern wir uns das

$$f'(x) = \lim_{x \to 0} \frac{f(x)}{x} \tag{165}$$

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

$$\lim_{x \to 0} \frac{g \circ f(x)}{x} = \lim_{x \to 0} \frac{g \circ f(x)}{f(x)} \frac{f(x)}{x} \tag{166}$$

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

$$\lim_{x \to 0} \frac{g(x)}{f(x)} = g'(f(x)) \tag{167}$$

$$\lim_{x \to 0} \frac{f(x)}{x} = f'(x) \tag{168}$$

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, $\overline{y}$ die übrigen Variablen. Es gilt:

$$\frac{d(f \circ g)}{dx} f \circ g(x, \overline{y}) = (\frac{df}{dx} f) \circ g(x, \overline{y}) \cdot \frac{dg}{dx} g(x, \overline{y}) \tag{169}$$

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

$$\frac{df}{dx}\frac{1}{1+e^{-x}} = \frac{e^x}{(e^x+1)^2} = \frac{1}{1+e^{-x}} \cdot (1 - \frac{1}{1+e^{-x}}) \tag{170}$$

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

$$S'(x) = S(x) \cdot (1 - S(x)) \tag{171}$$

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

15 Tiefe Netze trainieren

15.1 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 $D = \{(\vec{x}_1, y_1), ..., (\vec{x}_i, y_i)\}$
2. Unsere Kostenfunktion $K : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$.
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

$$(172)\quad \sum_{j=1}^{i} K(N_\theta(\vec{x}_j), y_j)$$

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:

$$(173)\quad N_\theta(\vec{x}) = g_{out} \circ M_k \circ g_{k-1} \circ M_{k-1} \circ ... \circ g_1 \circ M_1(\vec{x})$$

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

$$(174)\quad \underset{M_k, M_{k-1}, ..., M_1}{argmin} \sum_{j=1}^{i} K(g_{out} \circ M_k \circ g_{k-1} \circ M_{k-1} \circ ... \circ g_1 \circ M_1(\vec{x}_j), y_j)$$

Es ist natürlich klar, dass man für 174 keine analytische Lösung findet, es ist bereits schwer genug, einen Gradienten zu finden. Dafür fässt man diese Funktion einfach als eine Funktion auf, in der jeder Parameter eine Variable ist (also sehr hochdimensional!). Diese Funktion ist natürlich fast unmöglich sinnvoll darzustellen; man muss dazu die Variablen der Matrizen etc. linearisieren, Gruppen in *maxout*-units numerisch darstellen etc.

Wichtig ist das Konzept:

$\Rightarrow$ 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 berücksichigt die gesamte Batch. Die Größe der batch ist ein Hyperparameter. Also:

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

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

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

(176) $d(N_\theta(M), (\hat{y}_1, ..., \hat{y}_i))$

15.2 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.

$\Rightarrow$ 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

$$K(N_\theta(\vec{x}), \hat{y}) = F_{\hat{y}}(\vec{x}) \tag{177}$$

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

$$F_{\hat{y}} : \mathbb{R}^n \to \mathbb{R}_0^+ \tag{178}$$

ist eine positive Funktion (wegen K). Normalerweise (?) 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!

15.3 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 $\{(\vec{x}_i, \vec{y}_i) : i \in I\}$

$$F = \frac{1}{2} \sum_{i=1}^{n} \|N(\vec{x}_i) - \vec{y}_i\|_2 \tag{179}$$

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

Wir nehmen 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 $o \in \mathbb{R}$!

15.4 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:

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

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

15.5 Berechnungsgraphen

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

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

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

GRAFIK!

15.6 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:

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

Das bedeutet: falls 2 Knoten einen gemeinsamen Nachfolger in einer gewissen Tiefe haben, dann sind alle Nachfolger gemeinsam.

> Die einfache Variante von Backpropagation funktioniert nur für dichte Netze!

(Hier sind beispielsweise max-out units problematisch).

15.7 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** Funktion $r(v), l(v)$

1. $r(v) = f_v$
2. $l(v) = f'_v$

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

15.8 Backpropagation – Der Feedforward-Step

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

Fall 0 Die Eingabeknoten
Der Eingabknoten v_i bekommt

$$r(v_i) = x_i \tag{180}$$

$$l(v_i) = 1 \tag{181}$$

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

Wir kommen jetzt zu den inneren Knoten:

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

$$r(v) = f_v(r(\uparrow v)) \tag{182}$$

$$l(v) = f'_v(r(\uparrow v)) \tag{183}$$

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 : $f_v = x_1 + ... + x_i$

Der Knoten hat also eine Menge von Vorgängern.

$$r(v) = \sum_{v' \in \uparrow v} r(v') \tag{184}$$

$$l(v) = (1, ..., 1) \tag{185}$$

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

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

$$r(v) = F(\vec{x}) \tag{186}$$

$$l(v) = (1, ..., 1) \tag{187}$$

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.

15.9 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

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

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

(189) $l(v_o)$

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

> 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:

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

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:

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

Fall 3 $|\uparrow v| > 1$

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

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

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

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

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

15.10 Backpropagation: Ausgabe

Nachdem wir den FF und den BP-Schritt gemacht haben, haben wir für jeden Knoten v_i in unserem Berechnunggraphen 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 13 *Sei v_θ ein Knoten um Berechnunggraphen, 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.

15.11 Beweis von Satz 13

Induktion über die Struktur/Tiefe der Netze.

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

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

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

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

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

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

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

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

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

$$\uparrow^* n_1.$$

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

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

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

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

(Summenregel, für Ableitungen und Gradienten).

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

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

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

$f(\vec{x})$ 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

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

Dementsprechend ist nach Kettenregel

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

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

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

$$\begin{aligned} & g'(w_1 f_{o1}(\vec{x}) + ... + w_k f_{ok}(\vec{x})) \cdot w_1 f'_{o1}(\vec{x}) \\ & \qquad + ... \\ & + g'(w_1 f_{ok}(\vec{x}) + ... + w_k f_{ok}(\vec{x})) \cdot w_k f'_{ok}(\vec{x}) \\ = g'(w_1 f_{o1}(\vec{x}) + ... + w_k f_{ok}(\vec{x})) & \cdot (w_1 f'_{o1}(\vec{x}) + ... + w_k f'_{ok}(\vec{x})) \end{aligned} \tag{198}$$

   was genau einer Anwendung der Kettenregel entspricht.

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

15.12 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

(199) $\theta = M_1, ..., M_i$

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

(200) $\nabla_\theta K(N_\theta(\vec{x})\hat{y})$

Gradient descent macht nun einfach folgendes: wir setzen

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

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

15.13 Alternative: Momentum

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

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

Stattdessen soll aus wechselnden Richtungen eine generelle Richtung extrapoliert werden:

$$\vec{v}_0 := -l\nabla_\theta \frac{1}{m} K(N(\vec{x}), \hat{y})) \tag{202}$$

$$\vec{v}_{i+1} := \alpha\vec{v}_i - l\nabla_\theta \frac{1}{m} K(N_{\theta_i}(\vec{x}), \hat{y})) \tag{203}$$

$$\theta_{i+1} = \theta_i + \vec{v}_i \tag{204}$$

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 veringert 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!

15.14 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.

16 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 Intialisierung 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.

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

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 **dünner**.

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 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.

18 Word embeddings

18.1 Einleitung

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

Word embedding *Wir betten unsere Daten – also die Objekte, mit denen wir arbeiten wollen – in einen Vektorraum ein.* Das läuft unter dem Namen *word embedding*, und ist ein zentraler Schritt wann immer man Deep Learning auf linguistische Daten appliziert.

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

$$vec : \Sigma^* \to \mathbb{R}^n \tag{206}$$

$$vec : \Sigma \to \mathbb{R}^n \tag{207}$$

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. Wir betrachten eine Auswahl davon.

18.2 Vorspiel: tf-idf Kodierung

Eine alte Art der numerischen Gewichtung bzw. Kodierung (prä-neuronal sozusagen) von Worten

- tf steht für *term frequency* (ohne weiteres klar)
- idf steht für inverse document – die inverse der relativen Häufigkeit der Dokumente, die den Term enthalten.

Die Idee ist hier folgende: es gibt Worte, die sind relativ häufig *in allen Texten*: z.B. *a,the,...* etc. im Englischen. Diese Worte geben uns normalerweise keine relevante semantische Information über den Text. Es gibt auch Worte, die sind häufig *in bestimmten Arten* von Texten, aber nicht in allen. Diese Worte sind dann sehr wohl informativ für den Text.

Wir haben also das zugrundeliegende Muster: die Häufigkeit eines Wortes allein sagt nicht so viel aus wie seine Häufigkeit gewichtet mit der Anzahl der Texte, in denen es überhaupt vorkommt. Je mehr Texte das sind, desto geringer ist seine Aussagekraft – daher die Inversion.

Wir kommen zur Definition:

- Sei D ein Dokument, w ein Wort. Dann ist $tf(w, D) = |D|_w$ einfach die Zahl der Vorkommen von w in D
- Sei $\mathbf{D}$ eine Sammlung von N Dokumenten. Dann ist $idf(w, \mathbf{D}) = N/|\{D \in \mathbf{D} : tf(w, D) > 0\}|$. $\mathbf{D}$ lässt man dabei auch manchmal weg.

Dann haben wir

(208) $tf - idf(w, D) = tf(w, D) \cdot idf(w)$

Wir haben also eine Zahl, die einfach eine Heuristik der Informativität ist: je höher die

18.3 Skip-gram Methode

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:

Skip-gram Frage Kommen w, w' im selben n-gram vor?

Oder anders steht w' im Skip-gram von w?

Dieses Problem möchten wir approximieren mittels eines **linearen Modelles**.

Formaler, wir möchten den folgenden Term approximieren:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c\&j\neq 0} -logP(X_{t+j}=w'|X_t=w) \tag{209}$$

wobei

- c die Größe des Skip-gram Kontextes,
- T die Größe des Lexikons
- X_i das $i$te Wort ist
- w das Skip-Wort (in der Mitte)
- Die Wahrscheinlichkeiten P werden einfach vom Korpus geschätzt.

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

$$w_1\ w_2\ ...w_{10} \tag{210}$$

dann wird unser Datensatz:

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

wobei wir natürlich die Zahl der Vorkommen uns merken müssen.

Die Aufgabe: Finde eine Vektorrepäsentation der Worte

$$w_t : t \leq T$$

so dass ein Perzeptron P, gegeben den Eingabevektor $vec(w)$ für jedes w' die Wahrscheinlichkeit liefert, dass w' im Skip-gram vom 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

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

Dann nehmen wir

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

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

Gegeben $vec_1(w)$, finde für w' so dass

(212) $P_{skip-gram}(w'|w)$

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

Das bedeutet: wir möchten ein Modell trainieren, mittels Einheitsvektoren gut die syntaktische Nachbarschaft von Worten vorherzusagen. Dieses Modell ist im Normalfall ein einfaches **Perzeptron**, also eine Funktion der Form

(213) $g \circ M$

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

(214) $(g \circ M(vec_1(w_i)))_j \in [0, 1]$

ist eine Wahrscheinlichkeit, nämlich:

$\Rightarrow$ die Kookurrenzwahrscheinlichkeit $P(w_j|w_i)$

(Wichtig ist der skip-Teil, sonst wäre immer $P(w_i|w_i) = 1$)!

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

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

hat, ist

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

wobei m eine "vernünftige" Zahl ist (wie 300). Was nun passiert ist: $vec_1(w)$ *aktiviert* eine Spalte in M:

(215) $Mvec_1(w) = M_{-,i}$

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 einen Vektor als "Struktur" mit, und nehmen einfach am Ende diese Gewichte als Repräsentation – soz. Kategorie – des Wortes.

Das nichtlineare Layer Die Ausgabewahrscheinlichkeit

$$P(w|w')$$

berechnet man mittels der Softmax Funktion (die unter nichtlinear läuft). Aber Vorsicht:

$$M(vec_1(w)) \in \mathbb{R}^{300} \text{ (mehr oder weniger)},$$

das bedeutet: $softmax(Mvec_1 w) \in \mathbb{R}^{300}$ kann diese Wahrscheinlichkeiten *nicht* liefern

- wir brauchen ja eine Verteilung über $|Lex|$ Ereignisse
- und das kann leicht in der Größenordnung 10^5 sein!

(Man beachte übrigens das wir also die Wahrscheinlichkeit von $|Lex|^2$ Ereignissen schätzen müssen, als $(10^5)^2 = 10^{10}$. Dazu später.)

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

$$P_N(w'|w) = \frac{exp(Mvec_1(w')^\top Mvec_1(w))}{\sum_{v \in Lex} exp(Mvec_1(v)^\top Mvec_1(w))} \tag{216}$$

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

Dieses Modell wird – in diesem Falle – optimiert für M, d.b. für die eine Spalte in M, die durch $vec_1(w)$ aktiviert wird:

- Kostenfunktion ist, wie immer, Kreuzentropie
- Die "target-Funktion" ist die Skip-gram Wahrscheinlichkeit, unabhängig aus dem Korpus geschätzt (Maximum Likelihood)

Wir können nun mit den gewohnten Methoden das neuronale Netz trainieren; allerdings gibt es ein kleines Problem: wir haben oft

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

und die Kosten um

$$\nabla log(P_N(w|v)) \tag{217}$$

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 216 nur einmal (pro Trainingsrunde) berechnen müssen, müssen wir doch eine Vielzahl von Berechnungen ausführen.

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

$$\hat{P}(w|v) = \frac{\text{Anzahl der Vorkommen } (w, v) \text{ in den Daten}}{\text{Anzahl der Vorkommen } (x, v) \text{ in den Daten (beliebiges } x)} \tag{218}$$

Wir können daraus die Kostenfunktion ableiten:

(219) $K(w, v) = -logP_N(w|v) \cdot \hat{P}(w|v)$

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 $\hat{M}$ die Matrix unserer fertig optimierten Modells. Wir setzen nun (finale Einbettung):

(220) $vec(w) = \hat{M}vec_1(w)$

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

Die zugrundeliegende Annahme, die diese Einbettung rechtfertigt, ist folgende:

- unsere Vektorrepräsentation ist daraufhin optimiert, Kookkurenzen vorherzusagen,
- das bedeutet: diese Information (Kookkurrenz) steht irgendwo in diesen Zahlen – denn sonst könnte *Softmax* sie ja nicht herausholen.

Wir nehmen dazu 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.

18.4 Hierarchical Softmax: Die Optimierung

Wir haben oben das Problem erwähnt, dass wir in der Optimierung des Modells zuviele Rechenschritte ausführen müssen. Das effektive Training sprengt also in der Praxis oft unsere Möglichkeiten. Es gibt allerdings eine effiziente Art, die Berechnung oben 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

(221) $F : \Sigma^* \times \Sigma^* \to \mathbb{R}$

wobei

(222) $F(w, v) = -logP_N(w|v) \cdot \hat{P}(w|v)$

Wir nehmen also *zwei* Worte, wovon eines als gegeben, das andere als variabel betrachtet wird. Man kann auch eine Funktion definieren

(223) $F_w : \Sigma^* \to \mathbb{R}$

wobei

(224) $F_w(v) = -logP_N(w|v) \cdot \hat{P}(w|v)$

Wir können eine solche Funktion auch als einen **Baum** auffassen: wir legen eine erste Eingabe v fest; was fehlt ist die zweite Eingabe w. Wir nehmen nun einen **binären Baum**, wobei:

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

Hierbei ist folgendes wichtig:

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

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

$n(w, j) \cong$ ist der j-te Knoten auf dem Pfad von der Wurzel zu w

$L(w) \cong$ ist die Länge des Pfades von der Wurzel zu w

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

Für eine Gleichung $t = s$ gilt

$$\|t = s\| = \begin{cases} 1, \text{ falls } t = s \text{ wahr ist} \\ -1 \text{ andernfalls .} \end{cases} \tag{225}$$

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 $P_{HS}(w|v)$ nun wie folgt:

(226)

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

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

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

Aber wie verhält sich Softmax zu hierarchical Softmax?

18.5 word2vec und GloVe

word2vec und GloVe sind zwei existierene Embeddings, die auf den eben beschriebenen Methoden basieren: wir trainieren ein Perzeptron auf one-hot Vektoren für einen fake-task, nämlich Kookkurenz-Wahrscheinlichkeiten.

18.6 Sog. "Additive Kompositionalität"

Ein berühmtes praktisches Beispiel ist folgendes:

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

wobei $\approx$ bedeutet: es gibt kein $w \in Lex$ so dass $vec(w)$ näher an $vec(Paris)$ liegt als der Vektor in (227).

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

(228) $vec(kleines\ Auto) \approx vec(kleines) + vec(Auto)$

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

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

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

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

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

Wir haben (erinnern wir uns)

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

im rechtwinkligen Dreieck, also:

(232) $cos(0°) = 1$

Wir können also sagen:

> Zwei Vektoren $\vec{x}, \vec{y}$ sind sich näher (modulo Länge), je näher $cos(\angle(\vec{x}, \vec{y}))$ an 1 liegt.

Das lässt sich mittels der obigen Gleichung wiederum leicht ausrechnen (Termumformung). In diesem Fall interessiert also nur die Richtung des Vektors, nicht die Länge.

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

18.7 ELMo

Das Konzept (Embeddings from language models) Was wir bislang besprochen haben sind die klassischen Embeddings, die schon längst nicht mehr die besten Ergebnisse erzielen.

Elmo unterscheidet sich von den bisherigen Einbettungen in zwei entscheidenden Punkten:

1. die Einbettung eines Wortes wird nicht mehr an und für sich, sondern in einem gegebenen Kontext vorgenommen. Also: dasselbe Wort wird zu verschiedenen Vektoren, je nach dem Kontext in dem es steht.

2. Elmo basiert auf Buchstaben-embeddings: die Einbettung eines Wortes ist eine Funktion der Einbettung der Buchstaben. Das bedeutet: es gibt kein vordefiniertes Vokabular, alles kann eingebettet werden.

Ich glaube auch, das die beiden Punkt miteinander verwandt sind: dadurch, dass wir uns auf Buchstaben basieren, können wir besser *ad-hoc* einbetten. (Bei Worten gäbe es eine zu große Anzahl von möglichen Kontexten).

Der Vorteil der Methode liegt auf der Hand: wir können mit allen Arten von Ambiguität umgehen. Wenn wir die Einbettung als u.a. Semantik des Wortes betrachten, dann ist die Festlegung auf einen Vektor natürlich problematisch. Die kontextuelle Einbettung hat dagegen eine implizite Disambiguierung!

Der Nachteil der Methode liegt auch auf der Hand: wir brauchen viel Rechenleistung für die Einbettung, und insbesondere gibt es kein "fertiges Wörterbuch". Man kann also nur mit Elmo arbeiten, wenn man die entsprechende Rechenleistung zur Verfügung hat.

Einige Details Elmo macht die Einbettung auf Basis von Bi-LSTM, also rekurrenten Netzen. Die Idee ist folgende:

1. Ein Bi-LSTM wird vortrainiert auf einem großen Korpus für einen fake task (ohne Kosten kein Training). LSTM sind Netzwerke, die ihre Eingabe als **interne Zustände** kodieren (man kann sich das vorstellen wie einen Automaten).

2. Die Einbettung eines Wortes w in einem Kontext C ist eine Funktion des internen Zustandes des LSTM, nachdem es w, C gelesen hat.

Die Idee ist also ähnlich wie oben (Training für fake task), nur dass wir statt Perzeptron eine Technik benutzen, die auf Sequenzen arbeitet. Da wir LSTM noch nicht besprochen haben, macht es hier aber keinen Sinn in die Details zu gehen...

Einige Ergebnisse hierzu gibt es auf der Homepage des Allen Institute, die die Methode entwickelt haben: `https://allennlp.org/elmo`

18.8 BERT

(Bidirectional Encoder Representations from Transformers) BERT wurde von Google entwickelt und wird mittlerweile in den meisten Anwendungen benutzt. BERT ist gleichzeitig das state-of-the-art Modell. So wie Elmo auf LSTM basiert, basiert Bert auf dem Transformer-Modell, welches in den meisten Anwendungen die LSTM verdrängt hat.

Bert basiert auf einem **Sprachmodell**

(233) $LM : Lex^* \rightarrow [0, 1]$

also einem Modell welches die Wahrscheinlichkeit von Sätzen vorhersagen soll. Ein solches Modell kann man natürlich auch umgekehrt nutzen, um in einem gewissen Kontext das wahrscheinlichste Wort vorherzusagen:

(234) $C_{LM}(c_l, c_r) = argmax_{w \in Lex} c_l w c_r$

Der Transformer wird nun trainiert, indem wir einen Teil des Textes verdecken (ca. 15%), und versuchen die fehlenden Worte vorherzusagen. Für diesen task wird das Modell optimiert, und die entstehenden Parameter des Modells bilden die Basis des Wort-Kontext embeddings.

Man wendet bei BERT übrigens eine Art *fine-tuning*, auch genannt *transfer learning* an: das allgemeine Training erfolgt offline von Google, für spezielle Aufgaben wird BERT allerdings "nachtraininert" auf spezifischen Corpora.

19 Recurrent Neural Networks

19.1 Motivation

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

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

19.2 RNN – zwei Auffassungen

Mathematisch gesehen ist ein RNN eine Funktion

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

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

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

Wie wir das Netz auffassen, hängt von der Anwendung ab:

- falls wir ein Sprachmodell wollen, dann interessiert uns nur die letzte Ausgabe, die wir als Wahrscheinlichkeit auffassen;
- wenn wir ein Modell zur maschinellen Übersetzung nutzen möchten, dann interessieren uns auch alle intermediären Ausgaben.

Wir bezeichnen die beiden Modelle mit

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

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

19.3 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

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

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 $\vec{s_n}$.

GRAFIK!

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

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

$$RNN^+(\vec{x}_1, ..., \vec{x}_n, s_0) = (\vec{y}_1, ..., \vec{y}_n) \tag{235}$$
$$RNN(\vec{x}_1, ..., \vec{x}_n, s_0) = \vec{y}_n \tag{236}$$

wobei

$$\vec{y}_i = O(\vec{s}_i) \tag{237}$$
$$\vec{s}_i = N(\vec{s}_{i-1}, \vec{x}_i) \tag{238}$$

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:

$$s_4 = N(s_3, x_4) \tag{239}$$
$$= N(N(s_2, x_3), x_4) \tag{240}$$
$$= N(N(N(s_1, x_2), x_3), x_4) \tag{241}$$
$$= N(N(N(N(s_0, x_1), x_2), x_3), x_4) \tag{242}$$

Ebenso gilt:

$$
\begin{aligned}
y_4 &= O(N(s_3, x_4)) && (243)\\
&= O(N(N(s_2, x_3), x_4)) && (244)\\
&= O(N(N(N(s_1, x_2), x_3), x_4)) && (245)\\
&= O(N(N(N(N(s_0, x_1), x_2), x_3), x_4)) && (246)
\end{aligned}
$$

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!

19.4 Das *bag of words*-Modell CBOW

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

$$s_i = s_{i-1} + x_i \tag{247}$$

$$y_i = s_i \tag{248}$$

wobei

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

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

$$s_0 = (0, ..., 0) \tag{249}$$

Dann bekommen wir, für eine Eingabe

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

die Ausgabe

$$\vec{y}_m = \sum_{j=1}^{m} \vec{x}_j \tag{250}$$

Wie summieren also, im Falle einer Eingabe

$$w_1 ... w_n$$

einfach die Vektoren

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

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

19.5 SRNN

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

$$s_i = N_{SRNN}(x_i, s_{i-1}) = g(Ms_{i-1} + M'x_i + \vec{a}) \tag{251}$$

$$y_i = O_{SRNN}(s_i) = s_i \tag{252}$$

wobei

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

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

$$g = \frac{1}{1 + e^{-x}} \tag{253}$$

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.

19.6 RNNs stapeln

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

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

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

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

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

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

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

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

- Sei $N_1, ..., N_i$ die Funktionen des ersten,..., $i$ten Layers
- O die Ausgabefunktion

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

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

Dann ist also:

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

Desweiteren bekommen wir eine Liste von Zuständen

$$\vec{s}_1^{\,1} = N_1(\vec{x}_1, \vec{s}_0^{\,1}) \tag{255}$$

$$\vec{s}_1^{\,2} = N_2(N_1(\vec{x}_1, \vec{s}_0^{\,1}), \vec{s}_0^{\,2}) \tag{256}$$

$$\vdots \tag{257}$$

$$\vec{s}_1^{\,i} = N_i(...(N_2(N_1(\vec{x}_1, \vec{s}_0^{\,1}), \vec{s}_0^{\,2})...), \vec{s}_0^{\,i}) \tag{258}$$

Wir berechnen im nächsten Schritt also

$$\vec{y}_2 = O(N_i(...(N_1(\vec{x}_2, \vec{s}_1^{\,1})...), \vec{s}_1^{\,i})) \tag{259}$$

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.

19.7 Training von RNNs

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

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

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

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

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

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

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

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

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

$$F(N(\vec{x}_1, ..., \vec{x}_{n-1}), \vec{x}_n) \tag{260}$$

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

$$\frac{\partial F(N(\vec{x}_1, ..., \vec{x}_{n-1}), \vec{x}_n)}{N(\vec{x}_1, ..., \vec{x}_{n-1})} \tag{261}$$

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

$$\begin{matrix} \nabla_1 F(\theta_1, ..., \theta_i) \\ \vdots \\ \nabla_n F(\theta_1, ..., \theta_i) \end{matrix}$$

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

$$(\theta'_1, ..., \theta'_i) := (\theta_1, ..., \theta_i) - l \sum_{j=1}^{n} \nabla_j F(\theta_1, ..., \theta_i) \tag{262}$$

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

$$l = \frac{1}{n} l' \tag{263}$$

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

19.8 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.

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.

(264) $S(10) \approx S(10^{100})$

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

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

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

$$\vec{s}_0 \xRightarrow{\vec{x}_1} \vec{s}_1 \xRightarrow{\vec{x}_2} ... \xRightarrow{\vec{x}_n} \vec{s}_n$$

Das Problem ist: an jedem Zeitpunkt t_i hat unser Netz nur Zugriff auf den unmittelbar vorhergehenden Zustand und die neue Eingabe. Aber wohlgemerkt: das ist kein Problem für Berechnungen an sich, denn da der Zustandsraum unendlich ist, können wir immer irgenwie Informationen kodieren. Das Problem ergibt sich beim Training, wo gewisse Muster für unsere Augen "unsichtbar" bleiben, insbesondere Abhängigkeiten, die zwischen weit voneinander entfernten Eingaben bestehen. Was wir also suchen ist:

- eine Möglichkeit, am Zeitpunkt t_{i+j} unmittelbar auf Information in s_i zuzugreifen, ohne dass die Zwischenschritte einen Einfluss darauf haben.

Es geht also erstmal und allein um **Speicherzugang**. Einen solchen kontrollierten Speicherzugang kann man modellieren mit dem *Hadamard-Produkt* (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

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

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

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

Was geschieht hier? Eigentlich ganz einfach:

- das "gate" $\vec{g}$ nimmt sich diejenigen Komponenten von $\vec{x}_{i+1}$, die relevant sind (in einem zu bestimmenden Sinne);
- die anderen Komponenten sind die entsprechenden Komponenten von $\vec{s}_i$, also des vorigen Zustands (das macht $\mathbf{1} - \vec{g}$).

Das ist im Prinzip eine nur eine "Spielzeugarchitektur", denn wir führen eigentlich keine Berechnungen aus, wir entscheiden nur, welche Komponenten wir über welche Rechenschritte wir weitergeben, und welche wir erneuern; dennoch gibt diese Architektur ein Grundverständnis für was später kommt. Vor allem gibt es zwei Probleme hierbei:

1. Die gates sollen nicht nur Parameter aussuchen, sondern auch aktive Berechnungen ausführen.
2. Wir wollen die exakten gates nicht *apriori* (als Hyperparameter) spezifizieren, sondern als Teil des Trainingsprozesses optimieren. Das setzt aber voraus, dass sie **differenzierbar** sind, was bei $\vec{g} \in \{0,1\}^k$ nicht gegeben ist.

Zunächst lösen wir Problem 2, was deutlich einfacher ist: wir sagen einfach:

$$\text{nimm ein } \vec{g}' \in \mathbb{R}^k \text{ und setze } \vec{g} = S(\vec{g}')$$

wobei S die Sigmoid-Funktion ist. Auf diese Art wissen wir:

1. $\vec{g} \approx \vec{g}'' \in (0,1)^k$, und
2. die resultierende Funktion ist differenzierbar

Unser gate berechnet dann 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 addressiert genau dieses Problem. Die Idee ist, dass die Zustandsvektor $\vec{s}$ in zwei Hälften gespalten wird, nämlich

1. den (Langzeit)speicher $\vec{c}$, der Information (fast) unverändert über beliebige Strecken weiterreichen kann, wobei die Weitergabe durch gates (wie oben) kontrolliert und optimiert wird; und

2. den Arbeitsspeicher $\vec{h}$, der in jedem Rechenschritt komplett bearbeitet wird, wie im RNN

Wir definieren:

$$\vec{s}_{j+1} = (\vec{c}_{j+1}, \vec{h}_{j+1}) = N_{LSTM}(\vec{s}_j, \vec{x}_{j+1}) \tag{266}$$

wobei

$$\vec{c}_{j+1} = \vec{f} \odot \vec{c}_j + \vec{i} \odot \vec{z} \tag{267}$$

$$\vec{h}_{j+1} = \vec{o} \odot tanh(\vec{c}_{j+1}) \tag{268}$$

$$\vec{i} = S(M^{xi}\vec{x}_{j+1} + M^{hi}\vec{h}_j) \tag{269}$$

$$\vec{f} = S(M^{xf}\vec{x}_{j+1} + M^{hf}\vec{h}_j) \tag{270}$$

$$\vec{o} = S(M^{xo}\vec{x}_{j+1} + M^{ho}\vec{h}_j) \tag{271}$$

$$\vec{z} = tanh(M^{xz}\vec{x}_{j+1} + M^{hz}\vec{h}_j) \tag{272}$$

Außerdem wird die Ausgabe definiert als

$$\vec{y}_{j+1} = O_{LSTM}(\vec{s}_j) = \vec{h}_j \tag{273}$$

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

1. **i**nput,

2. **f**orget

3. **o**utput

Alle drei gates berechnen ein Perzeptron, also eine lineare Funktion $\circ$ 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 Arbeitspeicher kommt ($\vec{o} \odot tanh(\vec{c_j})$); gleichzeitig ist das ja der Output des Zeitschrittes.

$\vec{z}$ ist ein Kandidat, den (Langzeit)speicher zu aktualisieren; hier wird die $tanh$-Funktion verwendet. Das macht den Unterschied zu den gates:

- die gates sollen Werte in $[0, 1]$ annehmen (s.o.),

- während der eigentliche Arbeitsspeicher-Zustand $\vec{h}$ und der Update-Kandidat $\vec{z}$ in $[-1, 1]$ liegen. Daraus folgt:

- der Langzeit-Speicher $\vec{c}$ ist der *einzige* Vektor in der Architektur, dessen Komponente beliebige Werte in $\mathbb{R}$ annehmen können: hier werden Werte nicht normalisiert (mittels S oder $tanh$).

Das bedeutet, wir können uns wirklich Sachen merken, z.B. können wir mit (267) 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).

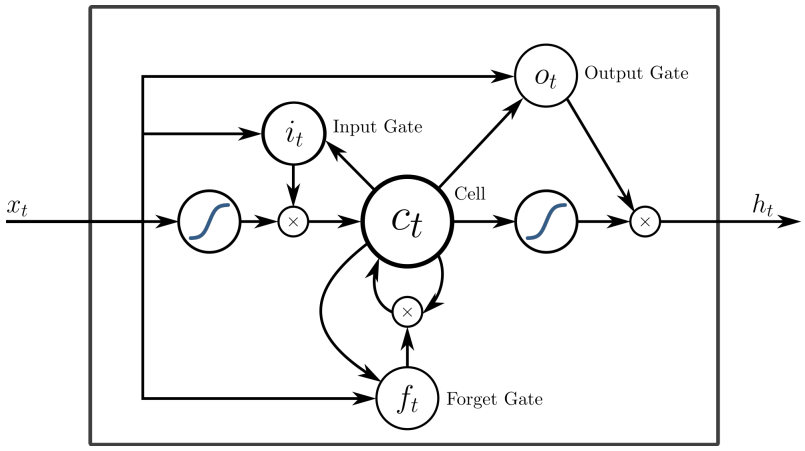
Wichtig ist es , zu verstehen was die Parameter des Netzes sind:

▶ 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.

Die Matrizen, die auf h_j appliziert werden, 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 Matrizen, die auf die Eingabe x_{j+1} appliziert werden, müssen natürlich dieselbe Ausgabedimension haben ($\cong$ Anzahl der Zeilen). Die Dimension der Eingabevektoren

ist davon unabhängig und wird im Normalfall reduziert durch die erste lineare Transformation.

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 lieferten bis vor einigen Jahren state-of-the-art Ergebnisse für die meisten Aufgaben, die das Modellieren von Sequenzen erfordern. Ein Problem dieser Architektur ist, das sie relativ kompliziert ist, und daher

1. die Ergebnisse schwer nachvollziehbar sind, und
2. das Training recht komplex.

Eine etwas einfachere Alternative liefern *gated recurrent units* (GRUs).

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:

$$\vec{h}_{j+1} = N_{GRU}(\vec{h}_j, \vec{x}_{j+1}) = (\mathbf{1} - \vec{z}) \odot \vec{h}_j + \vec{z} \odot \tilde{h}_{j+1} \tag{274}$$

wobei

$$\vec{z} = S(M^{xz}\vec{x}_{j+1} + M^{hz}\vec{h}_j) \tag{275}$$

$$\vec{r} = S(M^{xr}\vec{x}_{j+1} + M^{hr}\vec{h}_j) \tag{276}$$

$$\tilde{h}_{j+1} = tanh(M^{xh}\vec{x}_{j+1} + M^{hh}(\vec{h}_j \odot \vec{r})) \tag{277}$$

Der output ist gegeben durch

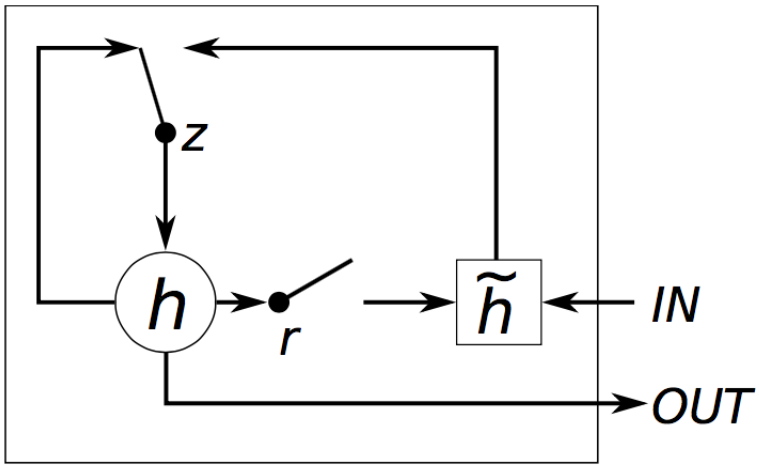
$$\vec{y}_{j+1} = O_{GRU}(\vec{h}_{j+1}) = \tilde{h}_{j+1} \tag{278}$$

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 $\vec{h}_j$
- $\vec{r}$ 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 Zwei Anwendungsbeispiele für RNN

Wir betrachten hier zwei Anwendungen für RNN (unabhängig von ihrer genauen Architektur!), 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.

21.1 RNN für *sequence labelling*

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

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

wobei gilt

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

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

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

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

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

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

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

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

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

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

softmax liefert eine Wahrscheinlichkeitsverteilung im Sinne, dass für

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

gilt:

$$(285)\ \sum_{i=1}^{n} softmax(\vec{x}) = 1$$

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

$$(286)\ label = \{l_1, ..., l_m\}$$

haben wir

$$(287)\ P_{RNN}(l_j|\vec{x}_1, ..., \vec{x}_n) = (softmax \circ RNN_1(\vec{x}, ..., \vec{x}_n))_j$$

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

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

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

setzen wir

$$(288)\ \hat{j} = \underset{m \in \{1,...,|label|\}}{argmax} (softmax \circ RNN_1(\vec{x}_1, ..., \vec{x}_j))_m$$

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

$$(289)\ RNN(\vec{x}_1, ..., \vec{x}_j) = label_{\hat{j}}$$

und zuletzt

$$(290)\ RNN^+(\vec{x}_1, ..., \vec{x}_i) = RNN(\vec{x}_1)...RNN((\vec{x}_1, ..., \vec{x}_i)$$

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

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

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

wir haben also am Ende Paare der Form

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

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

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

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

Wir berechnen die Kosten nun einfach mit der negativen log-Wahrscheinlichkeit (das entspricht *categorial cross-entropy*, wie immer!). Gegeben eine Paar der Länge n

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

gilt zunächst für das Präfix

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

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:

$$\begin{aligned}
K(x_1, y_1) &= -logP_{RNN}(y_1|vec(x_1)) \\
K(x_1x_2, y_1y_2) &= -logP_{RNN}(y_1|vec(x_1)) + (-logP_{RNN}(y_2|vec(x_1), vec(x_2))) \\
&\vdots \\
K(x_1...x_n, y_1...y_n) &= -\sum_{j=1}^{n} logP_{RNN}(y_j|vec(x_1), ..., vec(x_j))
\end{aligned}$$

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

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

21.2 RNNs als Sprachmodelle

Ein Sprachmodell ist eine diskrete Wahrscheinlichkeitsfunktion

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

bzw. $P : Lex^* \to [0, 1]$ (je nachdem ob man über Buchstaben oder Worte geht).

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 (die wahrscheinlichste Fortsetzung wird berechnet)
- ...

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 dass RNN, um, gegeben eine Eingabe

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

eine Wahrscheinlichkeitsverteilung zu induzieren, also:

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

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

(295) $$N((vec(w_1), vec(w_2), ..., vec(w_{n-1}), s_0) = s_{n-1}$$

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

Damit ist natürlich klar, was O ist:

(297) $O(\vec{x}) = softmax(M\vec{x})$

wobei $M \in \mathbb{R}^{|Lex| \times n}$, wobei n die innere Dimension des Modelles ist (also nicht extern bestimmt), die Ausgabedimension dagegen ist die Größe des Lexikons. Das ganze wird dann mit *softmax* in eine Wahrscheinlichkeitsverteilung transformiert. Also, falls das Lexikon entsprechend indiziert ist

$$Lex = \{w^1, ..., w^i\}$$

dann gilt

(298) $P_{RNN}(w^j|w_1, ..., w_{n-1}) = softmax(M(s_{n-1}))_j$

Hier ist es wichtig, den Satz-Index und den Lex-Index auseinanderzuhalten! 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

$$\begin{aligned} P_{RNN}(w_1...w_n) &= P(w_1) \cdot P(w_2|w_1) \cdot ... \cdot P(w_n|w_1...w_{n-1} \\ &= O(N(vec(w_1), s_0)_{j1} \cdot O(N(vec(w_2), s_1))_{j2} \cdot ... \cdot O(N(vec(w_n), s_{n-1}))_{jn-1}, \end{aligned}$$

wobei $j1, ...$ einfach die passenden Lex-Indizes sind.

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.

Das Problem des Lernens Aber: das allein besagt keinesfalls, dass Modell besser ist! Z.B. gibt es seit vielen Jahrzehnten PCFG und andere Modelle, die sehr mächtig sind, als Sprachmodelle in der Praxis aber völlig versagen. Warum? Das lässt sich mal wieder nur beantworten, wenn man das Training betrachtet, denn das Eintscheidende bei einem ML-System ist nicht was es *kann*, sondern was es *lernt*. Hier sieht man, dass die numerische Kodierung überlegen ist zu einer Kodierung mit Symbolen (zumindest in diesem Fall).

Wir trainiert man neuronale Sprachmodelle? Ganz einfach, Kosten sind die Abweichung von Vorhersage zu tatschlichem nchsten Wort, gemessen als Kreuzentropie. Man muss also keine zusätzlichen Annahmen machen!

Das Problem der Evaluation Aber mal eine grundlegende Frage: wir evaluiert man eigentlich ein Sprachmodell? Das ist alles andere als einfach! Denn es gibt ja kein "target", wo man accuracy, F-score, Pearson correlation oder etwas in der Art berechnen könnte! Menschliche Evaluation kann auch allerhöchstens bruchstückhaft gemacht werden. V.a. sind Menschen

keine Sprachmodelle, und haben keinerlei explizite Intuition darüber, wie "wahrscheinlich ein Satz ist" (falls das überhaupt etwas bedeutet). Was macht man also? Es gibt zwei Möglichkeiten:

1. Extrinsische Evaluation durch einen sog. "downstream task". Z.B. maschinelle Übersetzung (welches Sprachmodelle benötigt und aus verschiedenen Gründen besser evaluierbar ist. Das führt uns zuweit ab, wir werden das hier nicht betrachten.

2. Intrinsische Evaluation. Hierbei wird auf Eigenschaften des Modells geschaut. Wir werden eine betrachten.

Ein gängiges Werkzeug zur Evaluation von Sprachmodellen ist die sog. **Perplexität**: die Annahme ist, dass ein gutes Sprachmodell "reellen" Sätzen eine höhere Wahrscheinlichkeit zuweisen sollte als solche, die nicht "reell" vorkommen. Anders gesagt: je höher die Wahrscheinlichkeit, die ein Modell einem Datensatz zuweist (der natürlich ausreichend groß sein muss), desto besser sollte es sein, denn es verschwendet folglich weniger Wahrscheinlichkeitsmasse auf Sätze, die nicht vorkommen.

Das klingt nach Maximum Likelihood, und es ist damit auch verwandt. Allerdings gibt es zwei Dinge zu bedenken:

1. Wir suchen ein Mass, das eine Kostenfunktion abbildet (weniger $\cong$ besser)

2. Wir suchen ein Mass, das weitestgehend unabhängig ist von der Länge des Textes. Die Wahrscheinlichkeit sinkt aber immer monoton.

Beide Punkte sind schnell gelöst:

1. Wir nehmen den Kehrwert.

2. Wir ziehen die Wurzel daraus, und zwar mit Grad = Länge des Textes.

Und so kommt man von

$$P_N(T) \rightsquigarrow (\frac{1}{P_N(T)}))^{1/|T|} \tag{299}$$

wobei P_N die Sprachmodellwahrscheinlichkeit ist, T der Text auf dem evaluiert wird. Man beachte: die inverse Wahrscheinlichkeit ist > 1, also bedeutet die Wurzel, der Wert wird verringert. Das ganze nennt man dann die **Perplexität** des Sprachmodells (perplex so wie überrascht: geringe Wahrscheinlichkeit $\approx$ viel Überraschung). Also:

$$PP_N(T) = (\frac{1}{P_N(T)}))^{1/|T|} \tag{300}$$

Das ganze wird natürlich, wie immer in der Praxis, logarithmisch transformiert (aber nicht negativ, denn der Term ist > 1, also ist der logarithmus positiv!):

$$log((\frac{1}{P_N(T)}))^{1/|T|}) = 1/|T| \cdot log((\frac{1}{P_N(T)}))) \tag{301}$$

$$= 1/|T| \cdot (-logP_N(T)) \tag{302}$$

(Zur Erinnerung: man berechnet normalerweise Log-Wahrscheinlichkeiten, daher ist dieser Term besser zu berechnen!) Das also wäre die Perplexität. Weniger Perplexität heißt: das Modell ist besser.

22 Convolutional Neural Networks

22.1 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.

22.2 Convolution – Beispiel 1

Nehmen wir ein etwas blödes Beispiel aus der Literatur: wir haben einen **Laser-Sensor**, der uns sagen soll, wo ein **Raumschiff** ist; wir möchten also damit die Position bestimmen. Leider ist das Gerät fehleranfällig, von daher ist es sicherer, wenn wir über eine Reihe von Messungen mitteln. Allerdings kann es gut sein, dass das Raumschiff in der Zwischenzeit seine Position ändert. Wir haben also folgende Information:

- Unser Sensor liefert uns wechselnde Information über die Lokalität des Raumschiffs –
- wir wissen aber nicht, ob und zu welchem Grad diese Änderungen durch Messfehler oder Positionswechsel verursacht werden.

Was wir suchen ist die **aktuelle** Position des Raumschiffs. Aus obigen Gründen sollten wir nicht nur die aktuelle Messung berücksichtigen. Allerdings sollten wir die aktuelle Messung **stärker** berüchsichtigen als vergängene Messungen, anders gesagt: es gibt eine Funktion

(303) $w : \mathbb{R} \to [0, 1]$

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

(304) $s : \mathbb{R} \to \mathbb{R}^n$

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 $t \geq a$) mit

(305) $w(t - a)$

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

$$\mu(t) = \int_{0<a<t} s(a)w(t-a)da \tag{306}$$

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:

$$\int_{-\infty}^{\infty} w(x)dx = 1 \tag{307}$$

$$x < 0 \implies w(x) = 0 : \tag{308}$$

Wenn w für negative Werte ein positives Gewicht liefern würde, dann würden es uns (konzeptuell gesprochen) erlauben, in die Zukunft zu blicken.

Das unbestimmte Integral liefert nur die stetige Generalisierung der diskreten Funktion; wenn wir unsere Messungen in diskreten Abständen vornehmen (z.B. Sekunden), dann sieht das ganze wie folgt aus:

$$\mu(t) = \sum_{a=0}^{t} s(a)w(t-a) \tag{309}$$

Hier wird das Integral also eine (unendliche) Summe. Und genau diese beiden Operationen nennen wir **Konvolution**. Es geht hierbei also um eine **Gewichtung**; man benutzt hier das Symbol * und schreibt also:

$$\mu(t) = (s * w)(t) \tag{310}$$

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:

$$(311)\quad S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

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

$$(312)\quad (I * K)(i,j) = (K * I)(i,j)$$

was explizit bedeutet:

$$(313)\quad \sum_m \sum_n I(m,n)K(i-m, j-n) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

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

$$(314)\quad S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

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.

Nachtrag: einige realistischere Beispiele

Es gibt auch einige parallele Beispiele, die etwas nher am Alltag liegen. Ein gutes ist meiner Meinung nach die Reststrecke, die ein Auto fahren kann (in heutigen Autos wird das meistens angezeigt). Das ist natürlich eine Funktion von

- Benzinstand
- Verbrauch

Aber, beim letzten Punkt ist Vorsicht geboten: der "ewige" Durchschnittsverbrauch macht natürlich wenig Sinn (nimm an, jemand fährt seit Monaten nur in der Stadt). Der aktuelle Verbrauch noch weniger: jedes Anfahren und beschleunigen verbraucht enorm. Dementsprechend macht es Sinn, eine relativ flache Kurve mit Konvolution zu haben.

Ebenso übrigens die Schätzung von Fahrtzeiten aufgrund von aktuellem Verkehr (siehe google maps): je weiter das Stück entfernt ist, desto geringer ist die Rolle des (aktuellen) Verkehrs.

22.3 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).

(315) `Das Stück war lebhaft, aber nicht überzeugend.`

Hier sehen wir, dass es nicht genug ist, einzelne Worte zu betrachten – eine Betrachtung des Satzes als ganzes kann aber durchaus zuviel sein. Unser Modell basiert dagegen auf n-grammen, soll also jedes n-gram evaluieren (wir gehen davon aus, dass die Abhängigkeiten hinreichend lokal sind). Wir haben also erstmal, vor allem anderen,

Schritt 1: Word embedding Das haben wir besprochen; wir nehmen an, das

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

das word-embedding ist, $L \in \mathbb{N}$ eine beliebige Konstante, die wir uns merken.

Schritt 2: Die Eingaben Unsere Eingabe ist nun eine Sequenz von Vektoren

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

Unsere n-gram Größe legen wir fest auf k. Das bedeutet: wir definieren eine Operation auf

$$(\mathbb{R}^L)^k$$

Diese Operation wird ausgeführt der Reihe nach auf Vektoren

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

Wir operieren also der Reihe nach auf diesen Vektoren (von Vektoren).

Schritt 3: Die Convolution Wir definieren nun die Operation, die wir auf (#) ausführen. Zunächst definieren wir die Operation $\bigoplus$ mittels

$$\bigoplus(\vec{x}_1, ..., \vec{x}_k) = (x_1^1, ..., x_1^L, ..., x_k^1, ..., x_k^L) \tag{316}$$

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

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

Einfachheit halber setzen wir

$$\vec{y}_i = \bigoplus(\vec{x}_i, ..., \vec{x}_{i+k}) \tag{317}$$

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

$$g \circ M : \mathbb{R}^{L \cdot k} \to \mathbb{R}^j \tag{318}$$

Das bedeutet:

- $M \in \mathbb{R}^{j \times L \cdot k}$ ist eine Matrix;
- g ist eine nicht-lineare Funktion;
- dazu kommt noch ein **bias**-Term,

so dass wir einen neuen Vektor bekommen. Um das ganze übersichtlich zu machen schreiben wir

$$\vec{p}_i = g(M\vec{y}_i + \vec{a}) \tag{319}$$

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-grame selber spielt die Reihenfolge der Worte durchaus eine Rolle;
- ▷ Die Reihenfolge der k-grame untereinander spielt allerdings keine Rolle (bzw. höchstens als technisches Detail)

Das bedeutet als Zusammenfassung: die Konvolution erlaubt uns hier, aus der Gesamtinformation des Satzes gewisse Information zu behalten, während andere Information als irrelevant weggelassen wird. Wir können das also explizit im Modell (als Hyperparameter) spezifizieren – dann muss unser Modell das nicht mehr lernen, kann sich also auf andere Dinge konzentrieren.

Schritt 4: Pooling Was uns Convolution liefert ist ein Vektor von (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

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

(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:

$$mp : (\mathbb{R}^j)^n \to \mathbb{R}^j \tag{320}$$

was definiert ist durch die *max*-Operation:

$$(mp(\vec{y}_1, ..., \vec{y}_n))_m = \max_{1 \leq i \leq n}(y_1^m, ..., y_n^m) \tag{321}$$

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:

$$(ap(\vec{y}_1, ..., \vec{y}_n))_m = \frac{1}{n}\sum_{l=1}^{n}(\vec{y}_l)_m \tag{322}$$

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.

$$\begin{pmatrix} 1 & 5 & 8 \\ 6 & 9 & 0 \\ 4 & 6 & 1 \\ 9 & 2 & 4 \\ 1 & 7 & 3 \\ 0 & 3 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 9 & 8 \\ 6 & 7 & 7 \end{pmatrix} \tag{323}$$

wenn $k = 2$. Nun haben wir allerdings noch keinen Vektor, sondern eine Matrix. Wir machen daraus einen Vektor, indem wir die Vektor-komponenten konkatenieren:

$$(324)\quad \begin{pmatrix} 9 & 9 & 8 \\ 6 & 7 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 9 & 8 & 6 & 7 & 7 \end{pmatrix}$$

Eine andere Art von Pooling ist *dynamic pooling*. In diesem Fall werden die Vektoren zunächst in Gruppen aufgeteilt. Für jede Gruppe wird z.B. *max-pooling* appliziert; dann haben wir wieder eine Matrix (Liste von Vektoren), die wie im vorigen Fall auf einen Vektor reduziert wird. Dynamic pooling ist allerdings nicht ganz so interessant im allgemeinen Fall, da die Auswahl der Gruppen domänenspezifisches Wissen voraussetzt.

22.4 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 eine einzigen Eingabevektor der Länge $L \cdot n$ sehen, dann bedeutet die Konvolution, dass wir für jedes Fenster der Länge k dieselben Parameter anwenden. Man kann sich das konkret vorstellen als

- ein dünn verknüpftes layer (wir also eben nicht alle Zellen eines layers mit allen des nächsten verbinden)
- in dem viele Kanten dieselben Parameter haben.

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 $\vec{x}^1, ..., \vec{x}^n \Longrightarrow \vec{y}^1, ..., \vec{y}^n$, wie in POS-tagging

2. Eingabesequenz zu Vektor: $\vec{x}^1, ..., \vec{x}^n \Longrightarrow \vec{y}$, 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:

(325) $(\vec{x}^1, ..., \vec{x}^n)^T = (\vec{x}^n, ..., \vec{x}^1)$

Insbesondere bleiben die Zahlenvektoren selbst aber unverändert, also

(326) $(\vec{x})^T = \vec{x}$

Die Idee des bidirektionalen Netzes ist:

- Wir nehmen eine Eingabesequenz $(\vec{x}^1, ..., \vec{x}^n)$ und applizieren darauf *zwei* neuronale Netze.
    1. eines auf $(\vec{x}^1, ..., \vec{x}^n)$ selber,
    2. eines auf $(\vec{x}^1, ..., \vec{x}^n)^T$
- Die beiden Netze operieren unabhängig voneinander; erst im Ausgabelayer 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 ($\sim$ 2017–2019) 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. Die Gleichungend sind fehlerhaft, überarbeiten!!!) 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 Ausgagevektor o, Funktion von h, welcher eine Wahrscheinlichkeitsverteilung über Ausgabe y_{n+1} darstellt (also Nachfolger!)

Wir haben also in diesem Fall folgende Gleichungen:

$$\vec{h}_{j+1} = g(R\vec{x} + U\vec{o}_{j+1} + W\vec{h}_j) \tag{327}$$

$$\vec{o}_{j+1} = softmax(V\vec{h}_j) \tag{328}$$

$$\vec{L}_j = -log(\pi_{y_{j+1}}(\vec{o}_i)) \tag{329}$$

Hierbei muss man erklären: $\vec{o}$ wird gelesen als $P : Y \rightarrow [0, 1]$, wobei jeder

$$P(y) = \pi_y(\vec{o}) \tag{330}$$

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

$$N(\vec{x}, y_1, ..., y_n) = L_1, ..., L_n \tag{331}$$

Dann suchen wir:

$$\underset{\vec{y} \in Y*}{argmin} \sum N(\vec{x}, y_1, ..., y_n) \tag{332}$$

gegeben die Eingabe $\vec{x}$. Wir können das als eine Wahrscheinlichkeit auffassen, nämlich

$$\underset{\vec{y} \in Y*}{argmin} \sum N(\vec{x}, y_1, ..., y_n) \cong -logP(y_1, ..., y_n|\vec{x}) \tag{333}$$

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 beinflusst 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 $\Longrightarrow$ Vektor
- Vektor $\Longrightarrow$ Sequenz von Vektoren

Das ED Modell arbeitet nun, indem einfach die beiden Abbildungen **komponiert** werden:

$$\text{Eingabesequenz} \Longrightarrow \text{Vektor} \Longrightarrow \text{Ausgabesequenz}$$

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 denen 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 Eingabelnge 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.

"Traditionelles" SMT Um Transformer und Attention zu verstehen, macht es Sinn sich anzuschauen, wie statistische maschinelle Übersetzung "früher" (bis ~ 2015) funktioniert hat. Der normale Ansatz in statistischer maschineller Übersetzung war folgender:

- wir haben eine Alinierung zwischen Eingabe und Ausgabe, genauer gesagt eine Wahrscheinlichkeitsverteilung über Alinierungen.
- Um die Übersetzungswahrscheinlichkeit von Sätzen zu bestimmen, gewichten Übersetzungswahrscheinlichkeit von Worten mit der Wahrscheinlichkeit, dass sie aliniert sind.

  (Die Ausgabe wird dabei von links nach rechts konstruiert, aber jedes Wort kann dabei mit einem beliebigen Wort der Eingabe aliniert sein.)

- Wir suchen die Ausgabe, die die globale Wahrscheinlichkeit maximiert (Dekodierung).

  (Die Ausgabe muss übrigens noch durchs Sprachmodell, aber das läuft immer separat)

Das Transformer-Modell (von Goolge) möchte 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 bisheringen Ausgabe

Was aber ist Aufmerksamkeit?

Definition 14 *Aufmerksamkeit über eine Eingabe* $(\vec{x}^1, ..., \vec{x}^n)$ *ist eine gewichtete Summe*

$$\sum_{i=1}^{n} \lambda_i \vec{x}^i \text{ wobei } \lambda_i \text{ Skalar sind und } \sum_{i=1}^{n} \lambda_i = 1$$

Wir repräsentieren also eine Aufmerksamkeit über eine Eingabesequenz der Länge n mit einem Vektor $(\lambda_1, ..., \lambda_n)$*, wobei*

$$att[\lambda_1, ..., \lambda_n](\vec{x}^1, ..., \vec{x}^n) = \sum_{i=1}^{n} \lambda_i \vec{x}^i \tag{334}$$

Dieses Konzept, auch wenn es erstmal seltsam klingt, ist durchaus einleuchtend: ein Spezialfall von Aufmerksamkeit wäre die "one-hot"-Variante, sprich

$$att[0, ..., 1, , ..., 0] \tag{335}$$

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 $att_j = (\lambda_1^j, ..., \lambda_n^j)$

Nun gibt es folgende Korrespondenz:

(336) $\lambda_i^j \cong P(o_j \text{ ist aliniert mit } in_i)$

wobei o_j die $j$te Ausgabe ist, in_i die $i$te Eingabe.

Das liefert uns eine Intuition für Attention. Die Architektur des Modelles, der sog. **Transformer**, muss diese aber erst generieren, dazu kommen wir jetzt!

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. Wichtig: es handelt sich um ein Modell, welches eine Sequenz von Vektoren einliest. Dennoch ist es im gewissen Sinne ein VtS (Vector to Sequence) Modell, denn die *gesamte* Eingabe wird in *jedem* Rechenschritt gelesen. Gleichzeitig wird in jedem Schritt eine Ausgabe produziert, so dass wir die Korrespondenz haben

$$\text{Anzahl Rechenschritte} \cong \text{Länge der Ausgabe}$$

Insbesondere sind also Eingabelänge und Ausgabelänge unabhängig!

Die Eingabe Die Eingabe ist wie immer eine Sequenz von Vektoren

$$(\vec{x}^1, ..., \vec{x}^n),$$

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:

(337) $ann_i = (\vec{h}_i, \overleftarrow{h}_i)$

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:

$$e_{ij} = g(\vec{m}^\top(s_{i-1}, ann_j)) \tag{338}$$

$$e_{ij} \in \mathbb{R}\ (!) \tag{339}$$

Wir berechnen also in Schritt i immer n ($\cong$ Länge der Eingabe) verschiedene Alinierungen, nämlich für jede Annotation eine, während der Zustand konstant bleibt. 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:

$$att_i = softmax(e_{i1}, ..., e_{in}) \tag{340}$$

Wir definieren $att_{ij} = (att_i)_j$, das wäre also die oben beschrieben Wahrscheinlichkeit; also:

$$att_{ij} \equiv \lambda_j^i \tag{341}$$

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!

$$c_i = \sum_{j=1}^{n} \lambda_j^i ann_j \tag{342}$$

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:

(343) $s_i = f(s_{i-1}, y_{i-1}, c_i)$

(Laut meiner Quelle; hier ist y_{i-1} die vorherige Ausgabe) Hierbei ist f wohl ein einfaches Perzeptron, sprich:

(344) $s_i = g(M(s_{i-1}, y_{i-1}, c_i))$

Bleibt noch die Ausgabe y, die wir noch nicht besprochen haben:

Die Ausgabe Hier schweigt das Papier, ich schätze:

(345) $y_i = softmax(N(s_i))$

Damit können wir die Wahrscheinlichkeit jeder Ausgabe o ablesen, nämlich (wie gehabt)

(346) $L_i(o) = -log(\pi_o(y_i))$

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!

Zusammenfassung 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. Man muss also den Gradienten f.a. Komponenten finden!

24.4 Attention is all you need – der Transformer

Nach dem gleichnamigen Artikel (zu viele Autoren, sagen wir einfach 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?

$\Rightarrow$ Es gibt also keinen Zustand, der die bisherigen Berechnungen kodiert!

Stattdessen:

- 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*.

$$att(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{347}$$

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:

(348) $att(Q, K, V) = softmax(QK^T)V$

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 $softmax(QK^T)$ sind die Gewichte der einzelnen Value-Vektoren

Diese Attention layers werden also übereinander gestapelt. Zur Erinnerung: die Eingabelänge ist fix (z.B. 512), die Ausgabelänge wird bestimmt durch die Anzahl der Rechenschritte. Das erklärt, waru die Größe der obigen Matrizen **fix** ist.

Mir bleibt aber die ganze Architektur etwas unklar.

25 Neural Language Models, Sprachgenerierung

25.1 Sprachgenerierung

Sprachgenerierung basiert genau darauf: das Modell generiert immer genau das nächste Wort, welches es für das wahrscheinlichste gehalten hätte. Sprachgenerierung ist also relativ banal, wenn man ein Sprachmodell hat.

Das Problem ist meist ein anderes: neuronale Sprachmodelle brauchen Unmengen von Daten. Sprachgenerierung als *general purpose* ist ziemlich uninteressant; interessant sind spezielle Fälle:

- Chats
- Nachahmung bestimmter Autoren
- Bestimmte Anwendungen (medizinische Voruntersuchung, Nutzerhilfe, ...
- ...

Das Problem ist: gerade in diesen speziellen Anwendungen gibt es selten ausreichend Daten um *gute* Sprachgenerierung zu gewährleisten! Das macht **transfer learning** so interessant (s.u.)

26 ELMo, revisited

Motivation Nachdem wir also alle Zutaten beisammen haben, können wir jetzt "moderne" word embeddings betrachten, also solche die

- kontextabhängig sind
- und daher auf rekurrenten/Sequenz-basierten Modellen basieren

2022 ist Stand der Dinge: alle Erfolge der letzten Jahre basieren nicht auf ausgefeilteren Netzen zur Bearbeitung des tasks, sondern auf ausgefeilteren embeddings! Wir haben also guten Grund uns diese Sache genauer anzusehen. Die Motivation ist klar: jedes embedding sollte die semantischen und syntaktischen Eigenschaften eines Wortes auf die gewisse Art kodieren. Aber: diese Eigenschaften variieren, für dasselbe Wort, über Kontexte, da die meisten geläufigen Worte mehrere Lesarten haben. Das bedeutet, dass statische embeddings eben auch Ambiguität kodieren müssen, sowie die zugehörigen Mechanismen zur Disambiguierung. Dynamische embeddings lagern diese letzte Aufgabe aus, von daher müssen sie weniger leisten, und es ist nicht verwunderlich dass sie besser abschneiden.

Aufbau und Überblick Elmo basiert auf Bi-LSTM Sprachmodellen. Wir nutzen die inneren Zustände des Bi-LSTM, um unsere Wort-Vektoren zu generieren. Sprich, wir haben folgenden Arbeitsablauf für unser embedding:

1. Ein großes Bi-LSTM wird *offline* auf einem großen Korpus trainiert (wie üblich, indem man es verdeckte Worte vorhersagen lässt). Das Ergebnis nennen wir N_{Elmo}. Das ist das Pre-Training!
2. Wenn man nun die Worte in einem Text einbetten will, dann gibt man den Text als Eingabe in das Netz N_{Elmo}. Unser Netz hat nach jedem Wort a) einen inneren Zustand und b) eine Ausgabe. Man beachte: dadurch, dass es Bidirektional ist, enthält auch der Zustand des ersten Wortes Information über den gesamten Text!
3. Aus dem Zustand i generiert man $vec(w_i)$. Ganz einfach ist das aber nicht: N_{Elmo} ist **tief**, also ist der Zustand kein Vektor, sondern eine Matrix. Man muss daraus erst einen Vektor erzeugen (Linearkombination).

Details Ich folge [1] und nenne Bi-direktionale Sprachmodelle BiLM. Ein BiLM mit L layern hat, gegeben ein Eingabewort w (im einem gegebenen Kontext) $2L + 1$ Vektor-Represäntationen:

- L Repräsentationen im forward-Netz
- L Repräsentationen im backward-Netz
- 1 "Grundrepräsentation" (denn bevor wir ein BiLM aufs Wort applizieren, muss es ja schon ein Vektor sein!

Wir haben also eine Menge von Vektoren für ein Wort w_k in einem Satz $w_1...w_n$:

$$M(w_k) = \{\vec{h}^{\overrightarrow{LM}}_{k,j} : j \leq L\} \cup \{\vec{h}^{\overleftarrow{LM}:1\leq j\leq L}_{k,j} \cup \{\vec{x}_{w_j}\} \tag{349}$$

$\vec{x}_{w_j}$ ist hierbei die "naive" Einbettung von w_j, ansonsten haben wir die L Zustände des forward und des backward Netzes. Der Kontext ist hier soz. implizit im Sprachmodell, wird aber nicht explizit erwähnt. Wir setzen

$$\vec{h}^{LM}_{k,j} = (vech^{\overrightarrow{LM}}_{k,j}, vech^{\overleftarrow{LM}}_{k,j}) \tag{350}$$

$$\vec{h}^{LM}_{k,0} = \vec{x}_{w_j} \tag{351}$$

Das sind alles Dinge die sowieso anfallen. Jetzt kommt der eigentliche Elmo-Schritt: es gibt einen task-spezifischen Vektor $\vec{s}_{task} = (s_0, ..., s_L)$ wobei

$$sum^L_{i=1} s_i = 1 \tag{352}$$

sowie einen task-spezifischen Skalar $\gamma_{task} \in \mathbb{R}$. Etwas genauer, $\vec{s}_{task}$ ist ein softmax-normalisierter Vektor. Nun kommt die Definition:

$$ELMo(M(w_k), \gamma_{task}, \vec{s}_{task}) = \gamma_{task} \sum_{i=0}^{L} s_i \vec{h}^{LM}_{k,j} \tag{353}$$

Das liefert uns das embedding. Wie wir sehen basiert es auf den internen Zuständen des BiLSTM, und ist somit kontext-abhängig.

Es bleiben zwei Fragen:

1. Woher kommen $\gamma_{task}, \vec{s}_{task}$?
2. Woher kommt $\vec{x}_{w_j}$ (die naive Einbettung)?

Die erste Frage gleich im Anschluss, die zweite etwas weiter unten.

Der task-Vektor Es gibt zwei Extrembeispiele:

1. $\vec{s}_{task} = (0, 0, ..., 1)$. Also: nur das letzte (Ausgabe)Layer wird genommen! Das war eine beliebte Lösung vor Elmo [**?**].

2. $\vec{s}_{task} = (\frac{1}{L+1}, ..., \frac{1}{L+1})$. Das wäre das arithmetische Mittel.

Tatsächlich lässt sich empirisch zeigen, dass verschiedene layer fr verschiedene Aufgaben besser geeignet sind, also die entsprechende Information kodieren. Die unteren Layer kodieren eher Morphosyntax und Semantik, die operen Layer kodieren eher Information zu Lesart und Disambiguierung. Es gibt ein wichtiges Papier hierzu, allerdings für Bert (Bert rediscovers the linguistic pipeline).

Letztenendes ist es eine Frage der Erfahrung und der Evaluation. Allerdings ist $\vec{s}_{task}$ natürlich ein Hyperparameter und muss entsprechend "von Hand" optimiert werden. Das kann man entweder für einen konkreten Anwendungsfall selbst machen (Arbeit!), oder man kann sagen: für POS-tagging/Sentiment detection/... gibt es einen bewärten $\vec{s}_{task}$, den ich auch benutzen wollen, ohne Alternativen zu prüfen.

Elmo anwenden Die Anwendung von Elmo ist normalerweise wie folgt: nimm an, wir haben einen supervised NLP-task. Das bedeutet, wir haben für einen Text mit Worten $w_1, ..., w_n$ bereits zwei Dinge:

1. Ein (kontextunabhängiges) embedding $\vec{x}_1, ..., \vec{x}_n$

2. Ein entsprechender target-Datensatz Y, dessen Form start variieren kann (seq2vec, seq2seq,...)

Die Anwendung aus [1] besteht einfach darin, dass wir, für einen gegebenen task,

$$\vec{x}_i \mapsto (\vec{x}_i, ELMOM(w_k), \gamma_{task}, \vec{s}_{task}))$$

erweitern. Wir behalten also die ursprüngliche Repräsentation und erweitern sie mit Elmo. Y bleibt natürlich unverändert. Auf diesem Datensatz trainieren wir nun unser Modell (Achtung, veränderte Eingabedimension!). So einfach ist das! Elmo ist in diesem Sinne einfach ein **zusätzlicher preprocessing-Schritt**.

Die Archtektur des BiLM und $\vec{h}_0$ Wir haben noch nicht die konkrete Architektur des BiLM und die "naive" Einbettung besprochen. Natürlich ist das in gewissem Sinn variabel: man kann alles mögliche benutzen. In der Praxis benutzen aber die meisten das vorgefertigte "out of the box". Ich stelle hier die Architektur von [1] vor. Zur Erinnerung: es gibt *word*-embeddings und *character*-embeddings. Peters et al. nutzen character-embeddings.

Das **naive embedding** nutzt eine 2048-character n-gram convolution, gefolgt von zwei "highway layers" (Srivastava et al., 2015), und dann ein Perzeptron das die Dimension auf 512 reduziert. Wir sehen also, dass auch dieses character embedding bereits eine Menge Kontext beachtet, allerdings nur in einem konstanten Fenster.

Das **BiLM** ist ein 2 Layer Bi-LSTM (rekurrente Layer aufeinander gestapelt!). Die Eingabedimension entspricht dem naiven embedding, die Ausgabedimension ist ebenfalls 512. Wir haben also eine gewichtete Summe zweier Layer als Ergebnis des embeddings. Damit sind auch die task-Parameter überschaubar.

27 Bert: Transfer Learning

Bert, großes language model. Hieraus kann man viele kleine Aufgabe, wie POS-tagging, mit geringem Aufwand ableiten. Die Idee sehr einleuchtend:

1. Bert wird (von google) allgemein als language model trainiert, mit riesigen Datenmengen und großem Aufwand. Das ist der Ausgangspunkt

2. Mit einer viel kleineren Datenmenge wird Bert auf einen speziellen task "feinjustiert".

Intuitiv ist das klar, bei Menschen ist das auch nicht anders: wir können viele allgemeine Dinge, und auf dieser Basis lernen wir spezielle Aufgaben. Aber wie läuft das im formalen Modell?

28 Generative Adversarial Models

29 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:

$$(354)\quad acc(f, D) = \frac{\#\text{ falsche Klassifizierung von } f \text{ auf } D}{|D|}$$

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

$$(355)\quad prec(f, D, 1) = \frac{\#\text{ richtige Klassifizierung von 1 in } D}{\#\text{ Klassifizierung von 1 in } D}$$

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:

$$(356)\quad rec(f, D, 1) = \frac{\#\text{ richtige Klassifizierung von 1 in } D}{\#\ 1 \text{ in } D}$$

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:

$$F(f, D, 1) = 2 \cdot \frac{rec(f, D, 1) \cdot prec(f, D, 1)}{rec(f, D, 1) + prec(f, D, 1)} \tag{357}$$

Da Precision und Recall in $[0, 1]$ liegen, folgt dass der Zähler in $[0, 1]$ liegt; der Nenner in $[0, 2]$. Um diese inhärente Asymetrie auszugleichen wir 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.

30 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 fr 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. [**?**]!

31 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

$$\lambda \cdot (v_1, ..., v_n), \text{ wobei } \lambda \in \mathbb{R},$$

mit

(358) $\lambda \cdot (v_1, ..., v_n) = (\lambda v_1, ..., \lambda v_n)$

Die Addition zweier Vektoren ist definiert durch:

(359) $(v_1, ..., v_n) + (w_1, ..., w_n) = (v_1 + w_1, ..., v_n + w_n)$

Ein Vektorraum ist nun wie folgt definiert:

1. V ist abgeschlossen unter $+$ und $\cdot$ mit allen Skalaren.
2. V enthält 0_V, den sog. Nullvektor so dass $\vec{v} + 0_V = \vec{v}$.
3. V enthält für jeden Vektor $\vec{v}$ ein inverses Element $-\vec{v}$, so dass $\vec{v}+(-\vec{v}) = 0_V$. Natürlich ist leicht zu sehen dass $-\vec{v} = (-1) \cdot \vec{v}$.

Linear unabhängig Ein Vektor $\vec{v}$ ist **linear abhängig** von einer Menge $\{\vec{w}_1, ..., \vec{w}_i\}$, falls es $\lambda_1, ..., \lambda_i \in \mathbb{R}$ gibt so dass

(360) $\vec{v} = \lambda_1 \vec{w}_1 + ...\lambda_i \vec{w}_i$

Man nennt das eine **Linearkombination** von $\vec{w}_1, ..., \vec{w}_i$. Eine Menge $\{\vec{w}_1, ..., \vec{w}_i\}$ ist **linear unabhängig**, falls $\vec{w}_1$ nicht linear abhängig ist von $\{\vec{w}_2, ..., \vec{w}_i\}$ (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 spezifiert 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

(361) $p = \lambda_1\vec{r}_1 + \lambda_2\vec{r}_2 + \vec{s}$ 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 $\vec{r}_1, ..., \vec{r}_{n-1}$ (ungleich 0). Technisch gesehen ist die Hyperebene eine Menge von Punkten:

(362) $H = \{\vec{s} + \lambda_1\vec{r}_1 + ... + \lambda_{n-1}\vec{r}_{n-1} : \lambda_1, ..., \lambda_{n-1} \in \mathbb{R}\}$

Norm : definiert auf einem Vektorraum V ist das eine Funktion

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

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

1. $\|\vec{v}\| = 0 \Rightarrow \vec{v} = 0_V$
2. $\|\lambda \cdot \vec{v}\| = |\lambda| \cdot \|\vec{v}\|$, wobei $|\lambda|$ der Betrag ist
3. $\|\vec{v} + \vec{w}\| \leq \|\vec{v}\| + \|\vec{w}\|$

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

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

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

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

Für $p = 1$ vereinfacht sich das zu

$$\|(v_1, ..., v_n)\|_1 = \sum_{i=1}^{n} |v_i| \quad (363)$$

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

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:

$$\|(\sqrt{0.5}, \sqrt{0.5})\|_2 = \|(0, 1)\|_2 = 1 \quad (364)$$

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:

(365) $\|(\sqrt{0.5}, \sqrt{0.5})\|_1 = \sqrt{0.5} + \sqrt{0.5} > 1 = \|(0, 1)\|_1$

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

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

wobei natürlich

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

Es ist nicht schwer zu sehen dass gilt:

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

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

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

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

(Länge der Ankathete)/(Länge der Hypothenuse)

also $cos(90°) = 0$, $cos(0°) = 1$, und alles andere liegt dazwischen. Man kann den Kosinus auch verallgemeinert definieren als Funktion $cos : \mathbb{R} \to \mathbb{R}$, und zwar einigermaßen kompliziert als unendliche Reihe:

$$cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{2n!} \tag{369}$$

Was uns insbesondere interessiert ist der Kosinus zweier Vektoren. Im 2-dimensionalen Raum lässt sich das natürlich schön veranschaulichen in dem wir einfach einen Vektor als Hypothenuse auffassen, von seinem Ende eine Linie ziehen so dass sie den (verlängerten) anderen Vektor im rechten Winkel schneidet, und dann die geometrische Definition anwenden.

Allgemeiner (d.h. in beliebigen Dimensionen) lässt sich der Cosinus wie folgt berechnen: wir haben

$$\vec{x}^\top \vec{y} = \|\vec{x}\|_2 \|\vec{y}\|_2 cos\sphericalangle(\vec{x}, \vec{y}) \tag{370}$$

Daraus folgt mittels Termumformung:

$$cos\sphericalangle(\vec{x}, \vec{y}) = \frac{\vec{x}^\top \vec{y}}{\|\vec{x}\|_2 \|\vec{y}\|_2} \tag{371}$$

Skalarprodukt : Bislang haben wir Multiplikation nur mit Skalaren ausgeführt; es gab noch keine Möglichkeit, zwei Vektoren miteinander zu multiplizieren. Das macht das Skalarprodukt $\bullet : V^2 \to \mathbb{R}$. Wir multiplizieren also Vektoren und bekommen eine reelle Zahl (einen Skalar) als Ergebnis. Es gibt nicht das eine Skalarprodukt, sondern mehrere Arten es zu definieren. Wir brauchen hier nur die geläufigste, nämlich das Standard-Skalarprodukt:

$$(a_1, ..., a_n)^\top (b_1, ..., b_n) = a_1 b_1 + a_2 b_2 + ... + a_n b_n \tag{372}$$

Wenn wir Vektoren als spezielle Matrizen betrachten, ist das Standard-Skalarprodukt nur das Produkt zweier Matrizen, nämlich der Transposition der ersten und der zweiten. Wichtig ist dass wir die erste Matrix transponieren, sonst wäre das Produkt undefiniert; weiterhin setzt diese Multiplikation voraus, dass beide Vektoren die gleiche Länge haben.

Hadamard Produkt Das ist eine Funktion $\odot : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, einfach definiert durch **punktweise Multplikation**: $(x_1, ..., x_n) \odot (y_1, ..., y_n) = (x_1 y_1, ..., x_n y_n)$. 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 gewöhnungbedürftig; 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

$$(\mathbf{A} \cdot \mathbf{B})_{ij} = \textstyle\sum_{k=1}^{p} (A_{ik} \cdot B_{kj}).$$

Diese Multiplikation ist natürlich *nicht* kommutativ; um das zu sehen, nehmen wir an $p = 1$, Matrizen $\mathbf{A} \in \mathbb{R}^{3 \times 1}$, $\mathbf{B} \in \mathbb{R}^{1 \times 3}$; dann haben wir:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_2 & b_3 \end{pmatrix} = \begin{pmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & a_1 \cdot b_3 \\ a_2 \cdot b_1 & a_2 \cdot b_2 & a_2 \cdot b_3 \\ a_3 \cdot b_1 & a_3 \cdot b_2 & a_3 \cdot b_3 \end{pmatrix}$$

Wenn wir das umdrehen, dann bekommen wir:

$$\begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \big((a_1 \cdot b_1) + (a_2 \cdot b_2) + (a_3 \cdot b_3)\big)$$

Also bekommen wir in diesem Fall eine 1×1-dimensionale Matrix. Es kann natürlich auch oft vorkommen dass $\mathbf{A} \cdot \mathbf{B}$ definiert ist, während $\mathbf{A} \cdot \mathbf{B}$ undefiniert bleibt. Ein wichtiges Konzept in dieser Hinsicht ist die **Einheitsmatrix**, definiert als:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & \ddots & 0 \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

Diese Matrix ist also quadratisch, mit allen Werten 0, nur in der Diagonale hat sie den Wert 1. Einheitsmatrizen gibt es für jedes $n \in \mathbb{N}$, denn die Größe muss natürlich passen; wir denotieren die $n \times n$ Einheitsmatrix mit $\mathbf{1}_n$. Dann ist leicht zu sehen: Sei $\mathbf{A} \in \mathbb{R}^{m \times n}$ eine beliebige Matrix. Dann ist

$$\mathbf{A} \cdot 1_n = 1_m \cdot \mathbf{A} = \mathbf{A}$$

Eigenvektor und Eigenwert Der **Eigenvektor einer Matrix** M ist ein 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

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

sollten bekannt sein; intuitiv gibt der Wert der Ableitung $\frac{d}{d(x)} f$ an jedem Punkt den Grad an, in dem die Funktion zu- bzw. abnimmt. Wenn man nun eine multivariate Funktion hat, dann kann man sie nach jeder Variable

partiell ableiten. Sei $f : \mathbb{R}^n \to \mathbb{R}$ eine Funktion, $i \in \{1, ..., n\}$; dann haben wir mit

$$\frac{df}{dx_i} f(x_1, ..., x_n) \tag{373}$$

die Ableitung nach x_i, die uns angibt, wie die Steigung der Funktion in der i-ten Dimension verläuft; die anderen Variablen sind natürlich Parameter dieser Steigung: sie geben an wir groß sie ist an diesem Punkt im Raum. Geometrisch betrachtet: sei $f(x, y)$ eine Funktion, die für zwei Koordinaten einer Oberfläche die jeweilige Höhe liefert. Die Ableitung nach x gibt nun an, wie groß die Steigung ist, wenn ich in Richtung x gehe. Dasselbe Prinzip für höhere Dimensionen.

Der Gradient von $f : \mathbb{R}^n \to \mathbb{R}$ ist eine Funktion $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$, dessen Ausgabekomponenten 1-n durch die partiellen Ableitungen von f nach $x_1, ..., x_n$ berechnet werden, d.h.

$$\nabla f(x_1, ..., x_n) = (\frac{df}{dx_1} f(x_1, ..., x_n), ..., \frac{df}{dx_n} f(x_1, ..., x_n)) \tag{374}$$

Alternativ kann man auch die Einheitsvektoren zur Definition nutzen; nenne die n-dimensionalen Einheitsvektoren mit $i$ter Komponente 1 $1_1, ..., 1_n$; dann ist

$$\nabla f(x_1, ..., x_n) = \frac{df}{dx_1} f(x_1, ..., x_n) 1_1 + ... + \frac{df}{dx_n} f(x_1, ..., x_n) 1_n \tag{375}$$

Die geometrische Interpretation des Gradienten ist folgende: der Gradient einer Funktion gibt einen Vektor, und der zeigt in diejenige Richtung, in der die Funktion am schnellsten steigt. Also im Fall $f : \mathbb{R}^n \to \mathbb{R}$: die Richtung in der Ebene, in die man gehen muss um am schnellsten Höhe zu gewinnen.

Es ist klar, dass man, um ein *lokales* Maximum einer Funktion zu finden, einfach nur dem Gradienten "nachgehen" muss (mit einer gewissen *Lernrate* ϵ; wir haben das Maximum gefunden, falls der Gradient 0 an allen Stellen ist.

Lernrate Eine kleine Zahl $l > 0$, die besagt, wie wir den Wert der Funktion verändern. Was normalerweise wissen ist: wir kennen den Wert $f(x)$, und wir wissen dass für ein $\epsilon > 0$,

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

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) Perzetron Ein einfaches Perzeptron hat die Form

(376) $P(\vec{x}) = g(M\vec{x} + \vec{a})$

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. $softmax \circ ReLU$. Entscheidend ist dass es nur ein lineares layer gibt.

Multilayer Perzeptron (MLP), auch genannt feedforward network Ein MLP ist eine Funktion

(377) $MLP = P_1 \circ ... \circ P_n$

also eine Verkettung von Perzeptronen. Man nennt MLP auch feedforward Network; sie sind die einfachsten nichttrivialen neuronalen Netze. Das Beispiel aus 377 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] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.