Het leren van representaties voor symbolische sequenties

Representation Learning for Symbolic Sequences

Cedric De Boom

Promotoren: prof. dr. ir. B. Dhoedt, dr. ir. T. Demeester
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. B. Dhoedt
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2017 - 2018

ISBN 978-94-6355-116-8
NUR 984
Wettelijk depot: D/2018/10.500/34

Ghent University
Faculty of Engineering and Architecture
Department of Information Technology

imec
Internet Technology and Data Science Lab

Examination Board:

prof. dr. ir. B. Dhoedt	(advisor)
dr. ir. T. Demeester	(advisor)
prof. dr. ir. G. de Cooman	(chair)
dr. ir. T. Verbelen	(secretary)
dr. B. Coppens	
prof. dr. ir. T. De Bie	
dr. ir. S. Dieleman	
prof. dr. D. Van den Poel	

Supported by a Doctoral Fellowship of the
Research Foundation – Flanders (FWO)

Dissertation for acquiring the degree of
Doctor of Computer Science Engineering

Dankwoord

"Ik heb zelf wel eens gewonnen, 't liep nog vaker fout helaas. Succes is relatief en, ja, soms vangt een koe een haas. En is de neergang weer begonnen, dan foeter ik altijd: een winnaar is veel mooier als hij in een afgrond rijdt."

—Bart Peeters, 2010

Bij de start van elke nieuwe trein- en—ja, ik weet het—autorit lansheen de as tussen Gent-centrum en Beveren-'het Monaco aan de Schelde'-Waas, maak ik steevast een keuze over de soundtrack die de trip van de gepaste atmosfeer moet voorzien. Daarbij wordt zelden muziek gemeden uit het spectrum dat beslaan wordt tussen de vijfde symfonie van Prokofiev, *Graceland* van Paul Simon, de kleinkunstklanken van Buurman en het *Foute Uur* op Qmusic. Onlangs werd mijn aandacht echter gevestigd op een reeks podcasts onder de noemer *Freakonomics*. Iedere week word je als luisteraar opnieuw geprikkeld met vragen en inzichten rond economische, politieke, sociale en psychologische thema's. Eén van de meest beluisterde vertellingen uit 2017 droeg de welluidende titel *Why Is My Life So Hard?* en werd zelfs twee maal in datzelfde jaar uitgezonden. Het is veelzeggend dat een radioprogramma met deze titel zo populair geworden is, en tijdens het beluisteren ervan werd al snel duidelijk waarom. Uit recent onderzoek[1] blijkt namelijk dat we ons maar al te bewust zijn van de nadelen die we iedere dag ondervinden of van de situaties die ons doen en laten tegenwerken, maar dat we de voordelen, kansen, opportuniteiten, hulp van anderen, enz. vaak als vanzelfsprekend lijken te beschouwen. Dit effect wordt kortweg omschreven als de tegenwind-/meewind-asymmetrie: je herinnert je zeker wel nog een aantal situaties waarbij je sterk de wind van voren kreeg—letterlijk én figuurlijk—maar de keren waarbij het allemaal vanzelf lijkt te gaan, worden snel vergeten. In de academische wereld, en dan meer bepaald de levenssfeer van de doorsnee doctoraatsstudent, wordt het ervaren van extreme tegenwind al eens literair—met kleine 'l'—omschreven als de *valley of shit* of de *pit of despair*[2]. Ik geloof dat—maar corrigeer me gerust indien dit niet voor jou als lezer van toepassing is, hier

[1] Davidai, S., & Gilovich, T. (2016). The headwinds/tailwinds asymmetry: An availability bias in assessments of barriers and blessings. Journal of Personality and Social Psychology, 111(6), 835-851.

[2] https://oncirculation.com/2014/01/14/the-valley-of-shit-vs-the-pit-of-despair

is immers geen rigoureus empirisch experiment aan te pas gekomen—elke afstuderende doctoraatsstudent zich op z'n minst al eenmaal langsheen de rand van deze vallei begeven heeft, en dat sommigen er zelfs de bodem van verkend hebben. De truc is dan om voldoende kracht uit deze situaties te puren zodat je met een nieuw en herboren elan kan verdergaan, d.i. de tegenwind-/meewind-asymmetrie proberen omkeren. Want zoals Bart Peeters al aangaf in de beginquote van dit dankwoord: *"een winnaar is veel mooier als hij in een afgrond rijdt"*; met vallen en opstaan, zoals ze zeggen. Daarom is het van tijd tot tijd gepast om—en dit klinkt wel heel christelijk, alvast excuses hiervoor—stil te staan bij en dankbaar te zijn voor de toch wel luxueuze positie waarin je je bevindt, voor de zaken die je kan en mag verwezenlijken, voor de kansen die je te pas en te onpas in de schoot geworpen worden, voor de hulp—hoe miniem ook—die je uit elke ondenkbare hoek aangereikt wordt, voor de vriendschap en liefde die je mag ontvangen van en geven aan de mensen rondom je, enz. Ik zie dit dankwoord daarom niet als het zoveelste verplichte nummertje op de administratieve checklist bij het indienen van mijn doctoraatsproefschrift, maar eerder als een uitgelezen moment voor bezinning en—naar de mantra uit de specifieke lerarenopleiding—reflectie.

De vraag is dan natuurlijk hoe je hier juist aan moet beginnen. Er zijn altijd enkele personen die je maar al te graag in het lang en breed wil bedanken voor hun niet-aflatende steun tijdens de voorbije jaren. En er zijn uiteraard nog tien keer meer personen die je het leven aangenamer, makkelijker, gezelliger en interessanter hebben gemaakt, in welke vorm of kwantiteit dan ook. Uit de al dan niet ongegronde vrees voor potentiële diplomatieke en maatschappelijke rellen—"je gaat me toch wel in je dankwoord vermelden, hé!"—heb ik ervoor gekozen om het tekstuele aspect van deze dankbetuiging te beperken tot op het niveau van de sociale groep. Ik zal dus geen individuele namen vermelden, en al helemaal niet in volgorde van belangrijkheid, status, aantal *mutual friends*, genetische verwantschap en/of (materiële) sponsoring.

Vooreerst natuurlijk een grote portie dank naar mijn promotor-duo; jullie hebben mij de nodige kansen geboden om in alle nodige vrijheid onderzoek te verrichten in prikkelende, uitdagende en razend interessante materie. Ik kon steeds zonder schroom bij jullie terecht met de nodige vragen, problemen, opmerkingen, suggesties, paper drafts, enz. Daarnaast kreeg ik uitgebreid de gelegenheid om les te geven aan studenten uit de 1e en 3e bachelor en om thesissen te begeleiden van enkele laatstejaars, iets wat ik altijd met de volle goesting gedaan heb. Ik heb ook een mooi stukje van de wereld mogen zien door deel te nemen aan enkele internationale conferenties in Firenze, Atlantic City, Montréal, New York, Barcelona en Como. Een speciale vermelding ook voor mijn (ex-)bureaugenootjes uit het oude 2.22 in de Zuiderpoort en het nieuwe 200.026 in de iGent; bedankt voor de vele (intellectuele) discussies, grappen en grollen, uitdagingen, zelfbejammerend steen-en-been-beklag, enz. Uiteraard kon dit alles

ook niet zonder de nodige financiële steun vanuit de Vlaamse overheid en ook alle andere broodnodige ondersteuning vanuit onze fantastische vakgroep IBCN/IBCN+/IDLab (weet iemand het eigenlijk nog?) en uiteraard de UGent. Aan al deze mensen, van de kelder tot +12, zeer, zeer bedankt!

Ik neem ook meteen de gelegenheid beet om mijn doctoraatsjury te bedanken. Het lijkt me een verre van eenvoudige en moeiteloze taak om dit proefschrift kritisch en met de volle aandacht door te nemen, en om er nadien nog doordachte en zinnige vragen over te kunnen formuleren. Van mijn kant dus een enorm grote appreciatie voor jullie bereidwilligheid in het opnemen van deze taak.

Buiten de context van de universiteit, zou ik graag eerst en vooral mijn ouders en mijn hele gezin uit Beveren in de figuurlijke bloemetjes willen zetten. Ik werd altijd weer met open armen ontvangen, elke vakantie en elk weekend, en eigenlijk ook op iedere andere dag van de week, en nooit was een moeite te veel. Het gaf me ook de o zo nodige gelegenheid om even te ontsnappen aan de werkgedachte en het soms hectische en drukke stadsleven in Gent. De voorbije jaren hebben we samen nog heel wat mooie reizen gemaakt naar Italië, Noorwegen, Zuid-Afrika en binnenkort ook Canada, en ik hoop dat we dat in de toekomst nog veel zullen kunnen blijven doen! Ook aan mijn grootouders een welgemeende dank u voor alles wat jullie voor mij gedaan hebben, voor de interesse, de verwennerij, de bezorgheid en alle mogelijke vormen van steun!

Op dit punt zou ik graag enkele heel goede vrienden een speciale vermelding willen geven, maar ik besluit toch om niet af te wijken van mijn voornemen om geen namen expliciet te vernoemen. Zij die dit paragraafje nu lezen, zullen zich vast wel aangesproken voelen of weten over wie het gaat. Het zijn deze vrienden die samen met jou dezelfde weg afleggen—zij het ieder op zijn of haar eigen unieke manier—en waarmee je dus gelijklopende ervaringen, bekommernissen, succeservaringen, strubbelingen... kan delen, en dat schept zeker wel een sterke band. We zien of horen elkaar niet elke dag of iedere week, en dat hoeft ook niet, maar we weten dat we bij elkaar terecht kunnen om ons hart te luchten bij een chocomelk of Nespresso, pizza, babi pangang of Big Mac menu, of bij een spelletje Santorini, of gewoon een half uur lang op de hoek van een straat bij guur winterweer na een moordend saaie uiteenzetting over didactische bomen en muurtjes. Jullie vriendschap is onbetaalbaar!

In de categorie 'hier word ik reuzeblij van' denk ik ook met plezier terug aan de vele barbecues en weekendjes met ons groepje ex-computerwetenschappers, en ik hoop dat we deze tradities nog lang kunnen verderzetten. Ook in de muziekschool van Beveren ben ik ondertussen tot een deel van het decor verworden. Tot op de dag van vandaag krijg ik er de kans om pianolessen te blijven volgen en samen te spelen met talentvolle muzikanten. Een hele grote dankuwel aan de leerkrachten ginds—die me vaak al kennen van toen ik nog 1m30 was—het voltallige team, en natuurlijk alle andere mensen en vrienden die er voor zorgen dat de sfeer er al-

tijd top is! Om in de muzieksfeer te blijven, dankzij Harmonie Kunst en Vreugd uit Beveren heb ik vier jaar geleden het hobospelen terug opgenomen na een winterslaap van acht jaar, met als gevolg dat ik op vrijdagavonden vaak terug te vinden ben in het repetitielokaal onder de tribunes van het Freethielstadion. De repetities zijn dan wel vaak uitputtend, maar geven een mens toch veel voldoening na een harde werkweek. En als we er dan weer met trots kunnen staan op onze concerten twee of drie keer per jaar, dan mag je heus wel blij en dankbaar zijn dat je deel mag uitmaken van deze groep fijne muzikanten. Ook een zeer speciale bedanking voor de vriendjes in en rond Verkeerd Geparkeerd en de leuke activiteiten die we samen kunnen doen en organiseren—zwemmen in Van Eyck en Rozebroeken, film- en café-avonden, karting, lasershooten...—de meer dan toffe sfeer onder de medecursisten bij de specifieke lerarenopleiding aan het Kisp—want gedeelde frustraties scheppen een band, naar het schijnt—en de toffe weekendjes en zomerbarbecues met team Maasmechelen.

Ik sluit mijn dankwoord af met een bijzonder eerbetoon aan Spotify. In de zomer van 2016 kreeg ik namelijk de unieke kans om een kleine drie maanden naar New York te verhuizen voor een stage bij dit inmiddels beursgenoteerde bedrijf. Ik kwam er terecht in een enorm gedreven en dynamisch team en mocht mijn kennis over machine learning en deep learning botvieren op de populaire feature Discover Weekly; de neerslag hiervan vindt u terug in Hoofdstuk 6 van dit boek. Na de stage werden de contacten met Spotify aangehouden en heb ik mij het afgelopen jaar ingezet om samen met hen de RecSys Challenge 2018 te organiseren. Het was geen makkelijke klus om de Million Playlist Dataset samen te stellen en publiek beschikbaar te maken. Het was en is nog steeds een enorm plezier om met jullie samen te werken. Mijn uitdrukkelijke dank dus aan alle collega's, managers en vrienden binnen Spotify!

Ik heb—naar goede aangeleerde gewoontes tijdens de specifieke lerarenopleiding—ook een tweede versie van mijn reflectie gemaakt. Ik vermeldde eerder dat ik geen individuele namen zou vermelden, maar dit leek me achteraf niet toereikend genoeg voor een aantal bijzondere mensen. Daarom besloot ik om me even te concentreren en alle namen van deze personen en instanties op te schrijven zonder wie dit doctoraat niet mogelijk was geweest, zonder wie ik meer voor mezelf had moeten zorgen, zonder wie de afgelopen jaren minder aangenaam en interessant zouden zijn geweest—zowel binnen als buiten mijn doctoraat—zonder wie ik niemand had gehad om mijn onderzoek toch op zijn minst in grote lijnen aan uit te leggen, enz. Het resultaat is een wordcloud geworden waarin al deze namen werden opgenomen, in de vorm van een klavertje vier, waarover Wikipedia me vertelt dat *"volgens de legende elk deelblad iets voorstelt: het eerste hoop, het tweede vertrouwen, het derde liefde en het vierde geluk"*. Ik vermoed dat ik hiermee meteen ook het halve Nederlandstalige voornamenbestand van de Belgische staat gedekt heb, maar het is uiteraard de geste die telt.

Heel erg bedankt iedereen!

Gent, 22 mei 2018
Cedric De Boom

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
Annoy	Approximate Nearest Neighbors Oh Yeah
API	Application Programming Interface
AWS	Amazon Web Services

B

BM25	Okapi Best Matching 25
BPR	Bayesian Personalized Ranking
BPTT	Backpropagation Through Time
bWST	Baseline Short-Term Weight-based Model
bWLT	Baseline Long-Term Weight-based Model

C

CAT	Consensual Assessment Technique
CBoW	Continuous Bag-of-Words
CD	Compact Disc
CPU	Central Processing Unit
cuDNN	CUDA Deep Neural Network library

D

DBSCAN	Density-Based Spatial Clustering of Applications with Noise

E

EDCoW	Event Detection with Clustering of Wavelet-based Signals
EMI	Experts in Musical Intelligence

G

GHz	Gigahertz
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit

I

ID	Identifier
idf	Inverse Document Frequency

J

JS divergence	Jensen-Shannon Divergence

K

KL divergence	Kullback-Leibler Divergence

L

LDA	Latent Dirichlet Allocation
LSH	Locality-Sensitive Hashing
LSI	Latent Semantic Indexing
LSTM	Long Short-Term Memory
LT	Long Term

M

MIDI	Musical Instrument Digital Interface
MLP	Multi-Layer Perceptron
MP3	Moving Picture Experts Group Layer-3 Audio

N

NLP	Natural Language Processing
NMF	Non-negative Matrix Factorization
NP	Non-deterministic Polynomial-time

P

P	Precision; Probability
PCA	Principal Components Analysis

R

R	Recall
RAM	Random Access Memory
ReLU	Rectified Linear Unit
REST	REpresentational State Transfer
rHST	Recurrent Short-Term Listening History Model
rHLT	Recurrent Long-Term Listening History Model
RMSprop	Root Mean Square Propagation
RNN	Recurrent Neural Network
rPST	Recurrent Short-Term Playlist Model
rPLT	Recurrent Long-Term Playlist Model

S

ST	Short Term
SVD	Singular Value Decomposition
SVM	Support Vector Machine

T

tf	Term Frequency
tf-idf	Term Frequency Inverse Document Frequency
t-SNE	t-Distributed Stochastic Neighbor Embedding

W

WAV	Windows Wave

Samenvatting
– Summary in Dutch –

Als ik de term 'artificiële intelligentie' (kortweg: AI) laat vallen, denken velen onder ons spontaan aan een of andere futuristische robot. Dit beeld van de 'vleesgeworden cyborg' of trouwe personal assistant is niet eens zo gek, want het wordt ons voorgeschoteld in talrijke films waaronder *A.I.* (2001), *Blade Runner* (1982), *Ex Machina* (2015), *Her* (2014), *I, Robot* (2004), *WALL·E* (2008), *Avengers: Age of Ultron* (2015)... en tv-series zoals *Westworld* (2016) en *Black Mirror* (2011). Niet zelden wordt in deze rolprenten een ongemakkelijke, depressieve en zelfs dystopische maatschappij neergepoot. We komen er maar al te vaak terecht in werelden waarin robots en computers het dagelijkse leven beheersen en waarin de gewone mens is vervallen tot een nietszeggend wezen. Het negatieve beeld rond AI wordt daarnaast versterkt door de berichtgevingen in de lokale en internationale media. Zij smullen maar al te graag van het onheil dat ons te wachten staat, wat bangmakerij en vaak nietszeggende discussies over mogelijke doemscenario's in de hand werkt.

De voorbeelden hierboven reflecteren in veel gevallen een fictief toekomstbeeld en worden daarom vaak bestempeld als sciencefiction of fantasy. Waar de meesten onder ons echter niet bij stilstaan, is dat AI op dit eigenste moment al onder ons aanwezig is in de apparaten, applicaties, diensten... die wij op regelmatige basis gebruiken en raadplegen. Het bekendste voorbeeld is wellicht het weerbericht. Dagelijks worden ettelijke slimme modellen losgelaten op de weerkaarten, waarna zij proberen om het weer voor de komende dagen te voorspellen. De resultaten die uit deze algoritmen rollen, worden dan netjes opgeschoond en vervolgens gepresenteerd op tv om iets voor 8 uur 's avonds. AI is in deze zin niets anders dan een computeralgoritme dat voorspellingen kan doen en eventueel beslissingen kan nemen op basis van aangeleverde data. Een dergelijk algoritme hoeft niet eens zo complex te zijn: een eenvoudige thermostaat kan beslissen om te beginnen verwarmen wanneer de temperatuur onder 20 graden Celsius zakt, en slaat af wanneer de omgevingstemperatuur meer dan 22 graden bedraagt.

U ziet dus dat AI helemaal niet de angstaanjagende of mensachtige robot hoeft te zijn zoals het ons meestal wordt voorgehouden. Vaak zijn het

slechts kleine systeempjes die zich ongemerkt in de plooien of net onder de oppervlakte van onze maatschappij genesteld hebben. Uiteraard, er zijn altijd bedreigingen gepaard gegaan met elke grote ontdekking of wetenschappelijke vooruitgang in de maatschappij. En dat is voor AI en haar toepassingen niet anders. Denk bijvoorbeeld aan de algoritmen van Facebook en Twitter die je speciaal uitgekozen nieuwsberichten laten zien, en daarmee de polarisatie in de maatschappij ongewild in de hand werken. Of computers die autonoom handelen op internationale beurzen en als dusdanig op eigen houtje een beurskrach kunnen veroorzaken. En zogenaamde 'killer robots', drones die gemaakt zijn om een specifiek target te herkennen en uit te schakelen, zouden wel eens sneller realiteit kunnen worden dan verwacht.

Tot zover de maatschappijkritiek. Deze doctoraatsthesis gaat immers over het positieve verhaal van AI. Met name hoe AI ten dienste kan staan van de mensheid, van onze zelfontplooiing, en ter verbreding van onze eigen kennis en inzichten. We zullen het bijvoorbeeld hebben over hoe we relaties tussen woorden, concepten, entiteiten... kunnen achterhalen door gebruik te maken van de gigantische bergen informatie die iedere dag op het internet gegenereerd worden. Als we hier in slagen, zijn we bijvoorbeeld in staat om de zoekmachines van vandaag nog slimmer te maken, waardoor we eenvoudig informatie en allerhande gerelateerde content kunnen terugvinden. Naast de reële data die door gebruikers aangemaakt worden (Facebook posts, tweets, video's, foto's, ...), zullen we ook analyseren hoe deze gebruikers zich gedragen op bepaalde internetplatformen: welke producten koopt meneer x voornamelijk, naar welke video's kijkt mevrouw y in het weekend, etc. Als we in staat zijn om AI-toepassingen te maken die dergelijk gedrag anticiperen, kunnen we deze inzichten aanwenden om suggesties te geven aan de gebruikers, of ter ondersteuning van de marketingafdelingen, en dergelijke meer. Tot slot zullen we AI ook in een meer creatieve setting aan het werk zien. Door AI-systemen te maken die in staat zijn om zelf een muziekstuk of een melodie te componeren, kunnen we niet alleen het werk verlichten van professionele componisten en musici, maar krijgen we hopelijk ook extra inzichten in de bestaande muziektheorie.

In het volgende geef ik een korte technische samenvatting van de hoofdstukken in dit werk. De hoofdstukken bestrijken een variëteit aan AI-toepassingen die gebruik maken of gebaseerd zijn op zogenaamde symbolische sequentiële data. Met sequentiële data bedoelen we informatie die varieert in de tijd, en deze informatie zal voorgesteld worden als een reeks van symbolen: letters of woorden in een tekst, muzieknoten in een compositie, een stroom van tweets op Twitter, tracks in een playlist of album, etc. Voor elk van deze types data gaan we op zoek naar een AI-model dat in staat is deze data te interpreteren, samen te vatten, zelf te genereren, te anticiperen, te classificeren, enz.

In Hoofdstuk 2 beginnen we ons onderzoek op Twitter, waar we op zoek gaan naar tweets die dezelfde gebeurtenis ('event') beschrijven, en

waar we geconfronteerd worden met twee types sequentiële data. Enerzijds is er het sequentiële aspect van de Twitterstroom, waarop één na één duizenden tweets per seconden verschijnen. Als we deze data efficiënt willen verwerken, moeten de algoritmen om kunnen gaan met vluchtige data die niet zomaar kan worden opgeslagen. Hiervoor zullen we een online clusteringalgoritme gebruiken. Anderzijds bestaat elke tweet uit een opeenvolging van woorden, karakters, hashtags... In dit werk gebruiken we de standaard tf-idf-voorstelling voor tekst. Daarnaast trainen we ook een LDA-model (Latent Dirichlet Allocation) waardoor we ook de semantische betekenis van een tweet kunnen voorstellen, en we gebruiken hashtags om tweets te clusteren volgens semantiek. We zullen zien dat het gebruik van deze semantische voorstelling de prestaties van ons algoritme voor eventdetectie verbetert.

In Hoofdstuk 3 gaan we dieper in op tekstvoorstellingen. We zullen word2vec gebruiken als tool om woorden uit een groot tekstcorpus te projecteren in een laagdimensionale ruimte, wat resulteert in zogenaamde 'word embeddings'. We bekijken vervolgens hoe we de word embeddings uit een volledige zin kunnen combineren of aggregeren tot een 'sentence embedding' van hoge kwaliteit. Hiervoor richten we ons tot de document frequency van een woord als hoofdsignaal. In vergelijking met naieve aggregaties, komt onze methode als beste uit de bus op een dataset van zowel Wikipedia- en Twitterteksten. Het grote voordeel van onze methode is dat ze *out-of-the-box* bruikbaar is.

Voor we overgaan tot de laatste twee applicaties, heb ik een eerder theoretisch getint hoofdstuk toegevoegd. In Hoofdstuk 4 bespreken we namelijk in detail de werking van een recurrent neural netwerk voor het verwerken van karakters in een tekst, hoe dergelijke modellen getraind worden, en hoe we er data van kunnen samplen. We poneren drie trainingsprocedures en twee samplingmethodes, en geven algemeen advies wanneer welke methodes best gebruikt worden. Om tot deze claims te komen, hebben we gebruik gemaakt van vier datasets met tekst van verschillende aard.

In de laatste twee hoofdstukken bespreken we AI-applicaties die intensief gebruik maken van de kracht van recurrente neurale netwerken. Eerst bekijken we in Hoofdstuk 5 hoe deze modellen gebruikt kunnen worden voor het componeren van pianomuziek. Daarnaast creëren we ook een model waarin we rekening houden met de identiteit van verschillende componisten, en we observeren dat de gegenereerde muziek inderdaad verschillend is afhankelijk van de geselecteerde componist. We slagen er zelfs in om muziek te genereren waarin we de identiteit van de componist gradueel en naar believen kunnen aanpassen. In een luistertest met muzikale experts, ondervinden we dat het AI-model wel nog steeds het onderspit moet delven in vergelijking met echte composities.

Tenslotte zullen we in Hoofdstuk 6 de gebruikers op het online plaform voor muziekstreaming Spotify van naderbij onderzoeken. Meer be-

paald gebruiken we recurrent netwerken om het luistergedrag van deze gebruikers te modelleren. We ontwikkelen enkele modellen die in staat zijn om de beluisterde tracks van een gebruiker één voor één te verwerken, en vervolgens een voorspelling geven van de tracks die de gebruiker mogelijkerwijs in de toekomst zou kunnen beluisteren. Hiervoor maken we opnieuw gebruik van word2vec om de 6 miljoen meest populaire tracks op het platform te projecteren in een laagdimensionale ruimte. We tonen aan dat onze methode het beter doet dan standaardmodellen voor aanbevelingssystemen.

We besluiten dat, hoewel de ontwikkelingen in het veld van de artificiële intelligentie de laatste jaren al sterk toegenomen zijn, het onderzoek naar krachtigere modellen voor het verwerken van sequenties nog zal blijven groeien. In het laatste hoofdstuk formuleer ik in dat verband de hoofdzakelijke conclusies van dit werk en schets ik enkele onderzoeksvragen voor de toekomst. Het mag duidelijk zijn dat we het laatste van recurrent neurale netwerken nog lang niet gehoord hebben.

Summary

When I mention the term 'artificial intelligence' (in short: AI), many of us spontaneously think of some kind of futuristic robot. This image of the 'cyborg incarnate' or faithful personal assistant is far from surprising, because it is presented in numerous movies including *A.I.* (2001), *Blade Runner* (1982), *Ex Machina* (2015), *Her* (2014), *I, Robot* (2004), *WALL·E* (2008), *Avengers: Age of Ultron* (2015)... and television series such as *Westworld* (2016) and *Black Mirror* (2011). Not infrequently, an uneasy, depressive and even dystopian society is put down in these motion pictures. All too often we end up in worlds where robots and computers control daily life and in which ordinary people have become meaningless creatures. The negative image surrounding AI is strengthened by the reports in the local and international media as well. Newspapers are all too happy to tuck into the disasters that await us, which encourages intimidation and often meaningless discussions about possible doom scenarios.

In the examples above a fictitious vision of the future is often put down, and these stories are therefore labeled as science fiction or fantasy. Most of us do not realize, however, that AI is already among us in the devices, applications, services... that we use and consult on a regular basis. Perhaps the best-known example is the weather forecast. Daily, smart models are applied to various weather maps, after which they make a prediction for the weather situations in the coming days. The results that come from these algorithms are then neatly cleaned up and presented on tv a little before 8pm in the evening. In this sense, AI is nothing more than a computer algorithm that can make predictions and possibly make decisions based on provided data. Such an algorithm does not even have to be very complex: a simple thermostat can decide to start heating when the temperature drops below 20 degrees Celsius, and turns off when the ambient temperature is more than 22 degrees.

So, AI does not have to be the frightening or humanoid robot that is usually shown to us. Often they are just small systems that have settled unnoticed in the folds or just below the surface of our society. Of course, every major discovery or scientific progress in society is accompanied by threats. And this is no different for AI and its applications. Consider, for example, the algorithms of Facebook and Twitter that show you selected news items, and thereby unintentionally increase the polarization in soci-

ety. Or computers that autonomously trade on international stock markets and as such can cause a crash on their own. And so-called 'killer robots', drones that are made to recognize and eliminate a specific target, could become reality faster than expected.

So far the social criticism. This doctoral thesis is about the positive story of AI. In particular how AI can be at mankind's service, of our self-development, and to broaden our own knowledge and insights. For example, we will talk about how we can identify relationships between words, concepts, entities... by using the extensive amount of information that is generated on the internet every day. If we succeed, we will for example be able to make the search engines of today even smarter, so that we can easily find information and all sorts of related content. In addition to the real data created by users (Facebook posts, tweets, videos, photos, ...), we will also analyze how these users behave on certain internet platforms: what products does Mr. x usually buy, what videos does Mrs. y watch in the weekends, etc. If we are able to create AI applications that anticipate such behavior, we can use these insights to give suggestions to the users, or to support marketing departments in their decisions, etc. Finally, we will also put AI in a more creative setting. By creating AI systems that are capable of composing a piece of music or a melody, we can not only lighten the work of professional composers and musicians, but hopefully we will also gain additional insights in existing music theory.

In the rest of this summary, I will give a short technical summary of the chapters in this work. These chapters cover a variety of AI applications that use or are based on so-called symbolic sequential data. By sequential data we mean information that varies over time, and this information will be presented as a series of symbols: letters or words in a text, musical notes in a composition, tweets in the Twitter stream, tracks in a playlist or album, etc. For each of these data types we try to build an AI model that is able to interpret, summarize, generate, anticipate, classify, ... this data.

In Chapter 2, we start our research on the Twitter platform, where we look for tweets that describe the same event, and where we are confronted with two types of sequential data. On the one hand, there is the sequential aspect of the Twitter stream, on which thousands of tweets per second are appearing one after the other. If we want to process this data efficiently, our algorithms must be able to deal with volatile data that cannot easily be stored. For this purpose we will use an online clustering algorithm. On the other hand, every tweet consists of a succession of words, characters, hashtags... In this work we use the standard tf-idf representation for texts. In addition, we also train an LDA model (Latent Dirichlet Allocation) that captures the semantic meaning of a tweet, and we use hashtags to cluster tweets according to their semantics. We will observe that the use of semantic information improves the performance of our event detection algorithm.

In Chapter 3 we will go deeper into text representations. We will use word2vec as a tool to project words from a large text corpus into a low-dimensional space, resulting in so-called 'word embeddings'. We then examine how we can combine or aggregate the word embeddings from a complete sentence into one high-quality 'sentence embedding'. For this, we focus on the document frequency of a word as the main signal. In comparison with naive aggregation techniques, our method performs significantly better on both Wikipedia texts and tweets. The major advantage of our method is that it can be used out-of-the-box.

Before we proceed to the last two applications, we will focus first on the theoretical aspects of recurrent neural networks in Chapter 4. In this, we discuss in detail the training and sampling process for these kinds of deep networks. The main use case is modeling characters in a text. We present three training procedures and two sampling methods, and give general advice on when which methods are best used. To arrive at these claims, we have used four text datasets that each have a different nature.

In the last two chapters, we discuss AI applications that intensively use the power of recurrent neural networks. First, in Chapter 5, we investigate how these models can be used for composing piano music. In addition, we also create a model in which we take into account the style of different composers, and we observe that the generated music is indeed different depending on the selected composer. We even succeed in generating music in which we can adjust the style of the composer gradually throughout the piece. In a listening experiment with musical experts, we find that our AI model is still performing worse compared to man-made compositions.

Finally, in Chapter 6 we look at users on the online music streaming platform Spotify. More specifically, we use recurrent networks to model the listening behavior of these users. We develop a variety of models that are able to process the tracks a user has listened to one by one. The models are then able to predict future tracks the user might listen to. For this purpose, we use word2vec again to project the 6 million most popular tracks in a low-dimensional space. We show that our method outperforms standard models for recommendation systems.

We conclude that, although the developments in the field of artificial intelligence have increased considerably in the past few years, research into more powerful models for sequence processing will continue to grow. In the last chapter, I put the main conclusions of this work and I give an outline of a few research questions we can focus on in the future. It is clear that the part of recurrent neural networks in the story of AI is far from finished.

1 Introduction

"Fascination, fascination, it's just the way we feel. We love this exaltation, we want the new temptations, it's like a revelation, we live on fascination."

—Alphabeat, 2006

1.1 Life is a sequence

It might not immediately occur to you, but the text you are reading right now is composed of several patterns. These very words have been put in a specific order by the author of this doctoral thesis, so that you as a reader are able to attach a meaning to them. If alternative words had been chosen, or if the order of the words were scrambled, maybe the information conveyed in these sentences would have been different. Apart from the author's personal creativity, the arrangement of words is of course highly dictated by the grammatical rules of the language in which we are communicating, of which we all—whether or not unconsciously—have a basic understanding. For example, in English a noun can be preceded by an adjective (e.g. 'a brown fox'), while this is usually the other way round in Romance languages, such as French (e.g. 'un renard brun'). Everyone's internal linguistic alarm is immediately triggered when mistakes are made against such basic rules—and I hope I have not set anyone's alarm off yet! Our innate capacity of language learning and to instinctively sense how proper language should be structured, is called the *language instinct*, a term

coined by linguist Steven Pinker in his 1994 bestseller [1]. Building on the theories of Noam Chomsky, one of the brightest linguists and intellectuals, he claims that parts of the human brain are wired to reflect structures of a universal meta-grammar [2]. In the learning phase of childhood, these structures are heavily used to develop one's mother tongue.

On a more granular level, this text is also a sequence of characters from the Latin alphabet, next to spaces and punctuations. From very early on, we are taught to write words letter by letter—for the Flemish and Dutch readers, recall 'ik – maan – roos – vis'. This way we learn to associate spoken sounds with written symbols. That is, the order of the characters in the word reflects the order in which distinct sounds are made by the speech organ. Clearly, the words 'present' and 'serpent' use the exact same set of characters, but by changing the order, we arrive at different words and possibly different meanings—unless you get a pet snake for your next birthday. However, an interesting phenomenon is that, even though not entirely correct and not scientifically proven[1], it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteer be at the rghit pclae [3]. Even though this last sentence was perfectly understandable, you will not get hired easily or win a Nobel Prize if this becomes your standard way of writing. Next to this, a good illustration of heavy word 'corruption' in our daily lives can be found on social media platforms such as Twitter® and Facebook®, where it is commonplace to drop vowels, e.g. 'thnx' (thanks), 'plz' (please), 'k' (OK), 'ppl' (people), etc., to repeat them extensively, e.g. 'goooaaaal' (goal) or 'neeeeever' (never), or to undeliberately introduce typos, e.g. 'Egnlish' (English). These cases illustrate that humans can cope with dropping characters or changing the character order up to a certain extent, and that it does not necessarily imply a shift in the semantic meaning of the word or text.

In kindergarten and first grade, we also learn to separate words by spaces when writing full sentences. Of course, this is obvious to all of us, because after having learned to write spaces between words, no one ever questions it again. But there used to be a time when a teacher actually told you to do so, and you were reinforced to do it every time from then on, so that it became a natural thing to do. The same principle holds for writing periods at the end of a sentence, capital letters at the start of a sentence, commas before or after clauses, etc. Figure 1.1 shows an example of a didactic tool that is used in some schools for the purpose of writing spaces.

[1] For instance, 'a dootcr aimttded the maglteuansr of a tageene ceacnr pintaet who deid aetfr a hatospil durg blendur' is much more difficult to read; it translates to 'a doctor admitted the manslaughter of a teenage cancer patient who died after a hospital drug blunder'.

Figure 1.1: In some schools, a 'space man' is used to help children remember to write spaces between words. Source: www.teacherspayteachers.com.

As opposed to the language *instinct* that we naturally possess, spelling and writing rules are habits or *learned* behaviors, because we acquire them through instruction, exercise, punishment and reward. Since the rules for writing spaces and periods are generally simple, most people apply these rules consistently. But the writing rules for commas, verb conjugations, hyphenation... are more complex, less straightforward and often even not standardized. Some people have learned these rules by heart and have developed the habit of applying them with best effort, others have a vague understanding of the rules and merely follow their intuition, and yet others never even develop the habit of proper spelling and comma placement.

When we look beyond the character and word level, you might have noticed by now that this text is comprised of several sentences, one following after the other. The order of the sentences is important to effectively convey a story, an explanation, an argument, etc. At an even higher level, multiple paragraphs are arranged in a particular order, which in turn are grouped together in different sections and chapters. We might even look beyond this very thesis and consider for a moment all books we have read until today. Since books are usually read one after the other—or at least finished one after the other—the ordering of these books also forms a sequence. This sequence is a reflection of your reading behavior and shows which kinds of books your are mostly interested in, similar to how a sequence of sentences reflects the ideas and messages of the author.

The concepts of structured and ordered characters, words, sentences... are not just applicable to text. Actually, the world and human life is heavily organized in a sequential manner. These sequences can be anything, going from series of consecutive objects, numbers and patterns, to more abstract

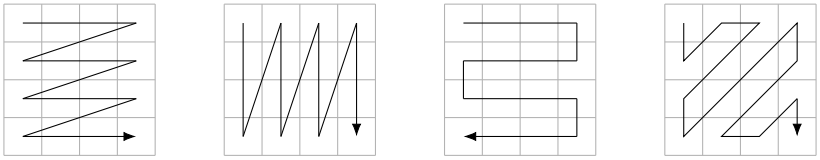


Figure 1.2: Four examples of how a picture can be interpreted and read as a sequence of pixels. From left to right: horizontal raster scan, vertical raster scan, horizontal snake scan and zig-zag scan.

orderings of states, situations, etc. Visual, *spatial* sequences are most easily observed: we have already covered the text in this book, but there are many more examples. For example, numerical sequences are abundant in telephone numbers, license plates, access codes... and the most interesting ones occur in IQ test questions such as 'complete the following sequence: 2, 3, 5, 9, 17, 33, ...' Other spatial sequences can be found in our DNA, composed of series of molecules that encode the biological mechanisms of life on earth. Sheet music is also a long chain of music notes, and is very similar to text in the sense that most songs are often grouped in phrases and parts, such as intro, verse, chorus, bridge, etc. Printed images can also be interpreted as sequences of individual pixels and color values, one positioned next to, above or underneath the other. An image can then be 'read' row per row or column per column, using traditional raster scan or 'snake' scan, etc. Examples of this are shown in Figure 1.2. Less obvious spatial sequences are observed on the road where cars are passing by one after the other. Bricks and other materials are piled on top of each other to construct a skyscraper. Pictures at an exhibition are often positioned to guide the visitor through an artist's story. And we conclude again with Facebook®, Twitter® and Instagram®, on which activity and news posts are selected and organized in a particular order to spike as much interest and interaction as possible.

A lot of sequences are not just orderings in space, but rather consist of events and phenomena that happen across time. In that case, we use the term *temporal* sequence. Time itself is the best example of a temporal sequence, and humans have structured it in terms of years, months, weeks, days, hours, minutes, seconds, milliseconds... To give another example, when we produce a sound with our voice, a pressure wave is initiated through the air. Such a wave travels in time through the air and can be regarded as a sequence of pressure values. Our eardrums are able to capture these pressure differences, and through the inner ear they are transformed so that our brain can interpret them again as sounds. If a sound wave is not captured or recorded, it is lost forever. An interesting observation is

that when a sound wave is recorded onto a physical medium as a series of pressure values—using the structure of an MP3, WAV, ... file—it becomes a spatial sequence! Under certain conditions[2], these spatial sequences can be transformed back into the original physical sound waves [4]. This is the basic principle behind digital sound recording and storage devices such as the compact disc (CD).

Another important example of a temporal sequence, and one that will play an important role in this doctoral thesis, is item consumption. All objects that you use, handle, touch, buy, rent, watch, listen to... during the day can be arranged in one very long series that is ordered in time. Again, this can be done on multiple abstraction levels. For example, when you make a cup of Nespresso® in the morning, the order of handled objects might be (cup, machine, capsule, machine, cup), but it can also be (cup ear, machine on switch, machine lid, capsule, machine lid, machine coffee button, machine lid, machine off switch, cup ear). In the current digital era, you should not be too surprised that even the smallest action you perform on your coffee machine is saved to a microchip, mostly for warranty purposes. In the context of this thesis, items will mostly be digital, non-physical objects. For example, all videos you have watched on YouTube® can be traced back to create one long sequence, which reflects your general interests and watching behavior, i.e. whether you have a short or long attention span by analyzing your skip behavior, how often you return to the same video over and over again, etc. These insights can then be used for targeted marketing purposes, to recommend additional videos to watch, to help users create videos that will reach the largest audience, etc. The same observation holds for other multimedia streaming services, such as Spotify®, Netflix®, but also for local cable companies (most notably Proximus® and Telenet® in Belgium). In a more literal sense, item consumption behavior is important for e-commerce websites such as Amazon®, Coolblue® and Bol.com®. For the owners of these websites, it is very important to gain insight into their customers, e.g. if someone buys a washing machine, he or she might also be in need of washing powder or clothespins. The website can also notice that you are probably looking for washing machines, and might come up with good suggestions of its own, or display what similar users have bought in the past. Being able to anticipate customer behavior can therefore greatly increase your sales. Unfortunately, these systems are far from flawless today; we can probably all recall the moment when we saw an online advertisement for the same shirt we had bought just yesterday.

[2]Since a physical (continuous) signal is transformed into a digital (discrete) signal, the Nyquist-Shannon sampling theorem has to be fulfilled in order to reconstruct the original physical signal.

1.2 Anticipation, expectation and surprise

In the past few pages I discussed how language, objects, sound, images, human actions, time... and, in fact, most of nature can often be regarded as sequences of small atomic entities, depending on your viewpoint. German-American philosopher Nicholas Rescher states that "natural existence consists in and is best understood in terms of processes rather than things – of modes of change rather than fixed stabilities" [5]. British sociologist Barbara Adam agrees and believes that "nature itself, the environment, and sustainability are no longer primarily seen through the lens of space and become fundamentally temporal realms, processes and concepts" [6, 7]. The fundamental parts of these processes and sequences are the past states, present states, future states, and the passage from one state to the next. *Causality* is an important and the most basic concept that drives a process, and describes how one state causes a change in a future state. Think for example of Newton's third law of motion, generally known as 'action-reaction', or the spontaneous reflex actions we perform to defend and protect ourselves, driven by our central nervous system.

However, human behavior and most of nature's processes cannot solely be explained by causality alone. In the first section of this introduction, we have already talked about the importance of being able to foresee what the future states of a sequence might look like. After all, our actions in daily life almost always have some associated goal towards the future. For example, we save or invest money on a regular basis because we want to live a happy life when we are old; we drive to the supermarket every week with the purpose of buying food, and we buy food in order to survive; to reach point B from point A, we put our left foot forward after our right foot (and vice versa) in order to walk; and in order to gain a profit, we invest in either stock X or stock Y. In Aristotle's principle of the four causes, this is best illustrated by the 'final' cause, also called telos (Gr. τέλος). It tries to formulate an answer to the question why and for which purpose things are what they are, because there is a final cause or end to be met. For example, animal teeth grow in the way they do for biting and chewing food, and not in some random way, which is good for the survival of the animal [8].

The reason why we can live a goal-oriented life, is due to the concept of *anticipation*. That is, we are able to continuously make decisions and perform actions based on our expectations of what the future would look like whether or not we were to proceed with our intentions. German economic sociologist Jens Beckert describes anticipation as 'fictional expectation', since "present imaginaries of future situations [...] provide orientation in decision making" [9]. The capability of projecting our thoughts into

the future is not just a feat of humanity, but is present in the whole of nature, going from biological ecosystems to physical processes [6]. Even on the tiniest micro-level this principle is at work: while driving on the highway, you won't suddenly brake since you expect the cars in front of you to keep on driving; at the same time, you keep a distance from the other cars in case one of them starts braking. While in bed at night, you will—hopefully—be peaceful and calm, because you expect the situation around you to remain as is for the next couple of minutes or hours; if you would suspect that the ceiling could fall down any second, you probably would run out of the house. And people are having multiple conversations on a daily basis with each other, because we are able to anticipate each other's feelings, and through this we try to invoke certain reactions and responses. In all these examples, the mind thinks about what might happen in the near future. In common situations, such as resting and driving and working, we aren't even aware of this mental process taking place; in more unusual or even endangering situations, you probably remember quick flashes or 'visions' of the possible scenarios that might occur. Psychologists call this effect 'mental time travel' [10]. Sociologist Markus Schulz summarizes as follows: "we all lead our daily lives based on innumerable assumptions about the future, short-term and long-term, small and large" [11].

In the light of the observations above, we can conclude that the human mind is a system that is highly anticipatory. American theoretical biologist Robert Rosen—one of the founders of anticipation studies—defined an anticipatory system as "a system containing a predictive model of itself and/or its environment which allows the system to change state at one instant in accord with the model's predictions pertaining to a later instant" [12]. In the definition of Rosen, the concept of a predictive *model* of the environment holds a central position. When we are born, we have absolutely no clue about what is going on in our environment: we don't know how to interpret it and we certainly don't know how to interact with it. But already very quickly, things start making sense. We recognize mum, dad, and other people in the family. Small objects can be grasped and thrown on the floor. When we are hungry, we cry until we get food. And a picture's size can be increased by pinch-and-zooming on the tablet, which is something you cannot do in printed magazines. Through this experience, we are steadily building a model in our brain of how the world around us works and how we should interact with it.

First, this model contains a wealth of information about all sorts of cognitive features. Recall for example the language instinct we talked about earlier, or think of our ability to recognize and manipulate objects in a three-dimensional world. Second, and most importantly in the context of

this thesis, we develop a model describing the mechanics of our world, about the consequences of actions and causes of observations. If we drop a glass of water, we know that gravity will certainly pull the glass towards the earth; the floor will most probably be wet, but it is not certain whether the glass will break or not—or at least, we as humans cannot calculate this in such a short period. We learn that, when riding a bike or driving a car, we need to turn the wheel counterclockwise in order to make a left turn. We also find out how to express specific types of feelings to invoke reactions, and how respond to our friends when they feel depressed. And if we would commit a crime, we might get caught and be sent to jail—and God will punish you, depending on your viewpoint. These examples show that we instinctively know the consequence of our actions before executing them by mentally 'unrolling' different scenarios into the future.

The very fact that we are able to do so, is thanks to our internal model of the world. No one will imagine a glass hovering in the air when suddenly dropped, because that is not how our world works, unless of course you are aboard the International Space Station. In some cases, however, our internal model of some aspects of the world or the mental depiction of the future is less straightforward. When playing a game such as chess, Connect Four® or Stratego®, you have to start thinking about strategies: what might happen if I perform this very move? How can I trick my opponent? What could happen within five moves from now? Or when we speak a foreign language that we have not yet mastered thoroughly, we often have to think of specific words or grammatical constructions. In cases where we observe something that is not in line with our mental image of the world's mechanics and dynamics, we call it *surprise*. And it is the notion of surprise that lies at the basis of creativity and emotions, but also accidents and impulsive acts. The Canadian professor David Huron suggests that human emotions are governed by five different response systems: reaction, tension, prediction, imagination, and appraisal [13]. We already recognize reaction, prediction and imagination as parts of our discourse until now. Tension is usually caused by uncertainty, and uncertainty is caused by events that do not entirely align with our expectations. Magicians use uncertainty and tension as a central element in their act, and musicians often play out-of-chord notes, preferably on weak beats, to invoke interest from the audience. We, as spectators or listeners, then use our imagination as well as possible to predict the outcome or continuation of the uncertain event, and it is at these moments that we can get carried away by great works of art. The skill of invoking surprise—and therefore emotions—at the right moments, through the right interplay of tension and resolution, is what sets great musicians, writers, painters, directors, chefs, presenters... apart from

average ones. But surprise can catch us at any moment in our lives: in the first days of winter, we all need to adjust our driving style to account for longer braking distances and more slippery roads; when stocks suddenly start plummeting, and there is no clear cause, people will start selling and buying impulsively and emotionally. And also our reflex system is entirely tuned to surprises: when we trip over an object, our equilibrium reflex is immediately activated.

1.3 Deterministic and stochastic sequences

When we think of sequences in a purely mathematical context, we often consider series of subsequent numbers. Perhaps the most famous sequence in the whole field of mathematics, is the Fibonacci sequence. The main property of the Fibonacci sequence is that one term in it is the sum of the previous two terms. If the first two terms are both equal to 1, we arrive at the following pattern: 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Other sequences are less famous, but are abundant in a lot of applications, such as square numbers (1, 4, 9, 16, 25, ...), triangle numbers (1, 3, 6, 10, 15, ...), on-off signals (1, -1, 1, -1, ...), etc. These sequences follow certain patterns and relations that can be described mathematically. An *explicit* formula defines every term u_n of a sequence as a function of its position n:

$$u_n = f(n),\ n \geq 1. \tag{1.1}$$

For example, the sequence of square numbers can be described by:

$$u_n = n^2, \tag{1.2}$$

since every term is the squared value of its position in the sequence. Following the formula, we have that $u_1 = 1$, $u_2 = 2^2 = 4$, $u_3 = 3^2 = 9$, etc. By contrast, a *recurrent* formula defines every term u_n as a function of one or more previous terms:

$$u_n = f(u_{n-1}, u_{n-2}, u_{n-3}, ...)\,, \tag{1.3}$$

given enough values to start or bootstrap the sequence[3]. If we consider the square numbers again, it can be verified that this sequence can be written according to the following recurrent relation:

$$u_n = (\sqrt{u_{n-1}} + 1)^2\,,\ \ u_1 = 1. \tag{1.4}$$

[3]While working on this chapter, I got intrigued by the question whether the sequence of sines $\sin(n)$ could be written as a recurrent formula. With a some external help, the research team and I eventually found that $u_n = \frac{\sin 2}{\sin 1} u_{n-1} - u_{n-2}$ (given $u_0 = 0$ and $u_1 = \sin 1$), which is a surprisingly simple formula!

Through this formula, we easily calculate that $u_2 = (\sqrt{1}+1)^2 = 2^2 = 4$, $u_3 = (\sqrt{4}+1)^2 = 3^2 = 9$, etc.

Using both types of formulas, a sequence is fully described. This means that at all times we have knowledge of the past, present and future terms. If we connect back to the previous paragraph, this means that we can completely anticipate the future of a sequence since it has fully predictable behavior: you can easily and precisely calculate the 100'th, 1000'th, ... term. For a recurrent definition, the concepts of 'mental time travel' and 'fictional expectation' can almost be taken literally, since we have to explicitly unroll the sequence to determine the future terms.

A sequence for which there is only one fixed possible term for every position, is called a *deterministic* sequence (Lat. determinare, i.e. to limit, to fix). The 10'th square number is always 100, and never 101, 99 or any other value. We have to point out that, although the terms in a deterministic sequence are fixed, not all deterministic sequences can be cast into a formula. For example, the sequence of subsequent prime numbers (2, 3, 5, 7, 11, 13, 17, ...) is infinite and deterministic, but cannot be cast into a single closed-form and computable formula—or, at least, no-one has ever found such a formula. And everything that has been written down, recorded, printed, etc. can also be seen as deterministic sequences, (usually) without an explicit mathematical definition.

Most realistic sequences, however, are not deterministic. When we look at all events in our daily lives that we have covered already in this introduction, you usually do not know what event will come next, in the near future, or in the very far future. Of course, you can enforce what words you will speak within a few seconds from now, what hand movements you will make, what food you will buy next, etc. But even these actions can become uncertain in the case of unforeseen events: you suddenly fall down a staircase, someone might ask you about something random, you come across an interesting item in the supermarket aisle, etc. In fact, for many sequences we have a lot of certain knowledge about past and present events, but almost no certain knowledge about the future events. Even a radio program that has been recorded and is now being aired, can get distorted by various external factors, such as noise, interference, broken hardware, etc. The very fact that these sequences have uncertain behavior, is the origin of the name *stochastic* sequence (Gr. στοχάζεσθαι, stokhazesthai, i.e. to aim at, to guess). Figure 1.3 shows how the stochastic nature of a sequence can impact the outcome of a process. It shows a deterministic sine sequence along with five similar processes that have been disturbed by (Gaussian) noise. We notice that, when the noise level is increased, we become more uncertain about the future outcomes of the process when time progresses.

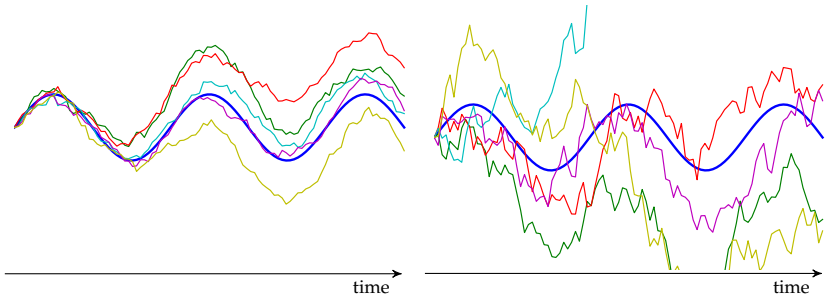


Figure 1.3: The blue line in both figures shows a standard sine wave as a process over time. The other five curves follow the same sine process, but at every time step a stochastic standard Gaussian noise factor is added. In the left figure a standard deviation σ of 0.1 is used; on the right, $\sigma = 0.4$.

In the right part of the figure, the cyan blue curve even disappears out of sight at the top of the graph.

As we have argued before, our brain is extremely well-designed and well-equipped to reason about and think of uncertain events in the future. This observation lies at the basis of the numerous actions we take on a daily basis. In fact, our brain is continuously processing multiple courses of actions, and it evaluates the outcomes of these courses. Sometimes we do this deliberately, when we think about our future selves or when taking important decisions. But most often these processes take place under the hood of our conscience. Of course, the number of actions that we can take and the associated consequences—the so-called *state space*, the set of all possible states—is extremely large, which renders the process of 'mental time travel' an extremely time- and resource-consuming job! But we, humans, can make unconscious decisions almost instantly, which suggests that the brain possesses a more intelligent system to handle future uncertainty, which is the 'model of the world' we have extensively talked about earlier. Sociologist and philosopher Roberto Poli puts it this way [6]: "When the brain takes a decision, it does not have sufficient time to traverse the state space of all the possible choices. To decide efficiently, the brain must decide which options are more likely to become real, i.e. it has to anticipate. [...] Apparently, neurons and more complex brain structures contain what have been called 'internal models' whose main task is to guide the system in its decision-making activities." These few sentences summarize and bring together the concepts of anticipation, decision making, the human brain, world models and stochastic sequences.

1.4 Artificial intelligence

Apart from some clues here and there, we do not know exactly how the brain stores, processes and reasons about its own model of the world. A lot of research has been done on this topic, and undoubtedly many researchers will continue tackling this very research question. After all, the idea that maybe one day we will be able to explain what mechanisms lie at the basis of human intelligence, is very tantalizing.

In the meantime, the field of *artificial intelligence*—in short: AI—has risen since the 1950s. Artificial intelligence is a research area lying on the intersection between mathematics, computer science, engineering, economics, biology, cognitive science, psychology, philosophy, and probably borrows from many other scientific fields as well. As the name already suggests, the main goal of AI is to try to mimic or even surpass genuine human intelligent behavior by means of artificial agents, i.e. computers, robots, phones, sensors, industrial machines, game consoles, etc.

A first observation we make, is that computers are traditionally very good at jobs that usually take a lot of effort for humans, such as calculating mathematical expressions and searching through large amounts of data. This is mainly thanks to the clever algorithms and ever-increasing computational power of these machines. On the one hand, our innate abilities of hearing, seeing, touching, speaking, ... are products of very long chains of evolution, since they lie at the basis of nature and all life on earth. Over time, our senses and brains have become highly optimized for these tasks, and it appears as if they take almost no effort. Dealing with and reasoning about complex mathematical problems, on the other hand, has only been around for ca. 5000 years, which is a mere blink in the history of evolution. It requires a lot of conscious brain power to solve such tasks, which explains why maths and sciences are often being perceived as tiring, overly complex, and abstract. To help us with such difficult and time-consuming problems, humans have invented different types of computing machines, of which the basic calculator and the modern-day computer are the best-known examples.

Secondly, we also observe that computers used to be bad at tasks that require human intuition, instinct and intelligence, such as recognizing cats in the street, playing games, interpreting texts, predicting market prizes, manipulating objects, etc. And this is were AI largely comes into play. Similar to how humans take decisions by considering visual, auditory and sensory input signals, artificial intelligence agents are designed to process specific types of input data, after which they perform an action for the task they are designed for. Over the course of the past 50 years, more or

less, researchers in various domains have developed a vast array of algorithms through which today's powerful computers, robots, machines... are able to achieve near- or above-human performance on a bunch of tasks. I have searched extensively for existing taxonomies of modern AI applications, but I was unable to find one that presented both an adequate and complete picture of the research field. The following taxonomy is therefore largely based on my personal point of view; and, as is the case with many taxonomies, other people might have approached it differently. It is not my intention to give a complete overview of the recent and early AI developments—for which I refer the reader of this thesis to Stuart Russell's reference work [14]—but for every category I will give a few well-known examples that have been important, groundbreaking or just interesting in the development of the field.

Cognitive applications

The AI applications in this first category mostly try to mimic human cognitive behavior. This means processing signals by sensing the environment, attaching a meaning to these signals, and finally being able to extract useful information and knowledge from the processed signals. The national postal services use such AI systems to sort letters based on their destination, since they are able to recognize and interpret handwritten addresses. Next to this, most of the modern car manufacturers implement traffic sign recognition as part of their on-board systems. The cars continuously take visual input from the road through a camera, extract the traffic signs and interpret the symbols. This information is then used to indicate whether the driver is driving too fast, or is automatically uploaded to the cloud to update real-life traffic conditions. As a final example, I include Shazam®, a mobile app that is able to recognize the song that is currently playing in the room around you. The way this app works, in short, is by matching small parts of audio signals with entries in a large catalog of most commercially available records [15].

Predictive applications

In a second category, I consider the AI applications that use current and historical data to make predictions about the present and the future—although this category of applications can also be used to research phenomena in the past. For example, a lot of startups such as Kayak® and Hopper® are nowadays tackling the task of predicting flight prices as accurately as possible. Suppose that you want to fly from Brussels to Vancouver this summer, these AI applications will use historical data as well as current sales

trends to give the user advice on when and how to book their flights in order to pay as little as possible. Also in the realm of transportation, the mobile app Waze® has become increasingly popular in the past few years. It relies on historical data, real-time crowdsourcing and predictive analysis to guide car drivers as quickly as possible to their destination. Finally, I present a broad category of examples that will turn out to be important in the remainder of this thesis, which is the realm of *recommender systems*. Such systems form an important part of every major service that deals with users and large item catalogs, such as (online) retailers (Amazon®, Zalando®, ...), streaming services (Netflix®, Spotify®, ...), libraries, social media, tourism (TripAdvisor®, Yelp®, ...), dating apps, etc. The goal of recommender systems, as the name suggests, is to present the user with a list of items (videos, food, music tracks, clothes, potential dates, ...) that he or she might be interested in trying next or in the future. In that sense, a recommender system is anticipatory towards a user's behavior in the future[4].

Active applications

AI applications that autonomously perform actions in their (digital or real-world) environment combine both cognitive and predictive properties. After all, such intelligent agents have to be able to interpret what they observe in order to choose their next actions. In this category, we have the numerous AI bots that are able to play games against humans. The first well-known examples are chess computers. Already at the end of the 18th century, the Hungarian inventor Wolfgang von Kempelen built a chess robot. It was, however, a fake robot, because there was actually a person sitting in the cabinet underneath the chess board. In 1996, IBM® made the first computer, 'Deep Blue', that was able to beat a chess world champion under standard time controls: it defeated Garry Kasparov in 1997, although this is disputed [16]. Deep Blue later resulted in the development of IBM's famous AI computer named Watson, which was able to defeat human players in the television game Jeopardy! From 2014 on, Alphabet's DeepMind has recently pioneered in building AI agents that are able to play video games, and their algorithms have defeated professional players in the ancient game of Go [17, 18]. In this category of AI applications, we also find physical robots, both industrial and humanoid. To pick one major feat at the end of 2017, the famous robotics company Boston Dynamics made their robot Atlas do a somersault—although their AI is largely based on rule-based engineering and control theory.

[4]These systems are so powerful nowadays, that in 2012 the data company Target was able to predict a teenager's pregnancy based on her shopping list without her family knowing. Her father found out after the supermarket had sent coupons for baby clothes and cribs.

Generative applications

In the final category, I consider those applications that are able to generate new data themselves, while the applications above mainly use existing, user-provided data. A well-known example are the computer generated voices in navigation systems and digital assistant such as Siri® (Apple®), Alexa® (Amazon®) and Cortana® (Microsoft®). Recently, intelligent chatbots have been deployed on a number of websites. For example, at the end of 2017, the Ghent-based company deJuristen even launched a prototype of their chatbot that gives legal advice; their ultimate goal is to render traditional lawyers obsolete. Finally, there also exist numerous applications in the creative domain, in which AI is used to compose music [19, 20], draw paintings [21], or even create new AI systems themselves [22]!

Most of the applications listed above are designed to make our lives easier and more pleasant. They also help us at tasks that are difficult, tedious and time-consuming, such as finding out the artist and title of that one song that is now playing on the radio. And, especially for the applications in the creative and active domain, they teach us a lot about how humans approach and think about certain tasks. What all applications have in common, is that they possess a model of the problem we are trying to solve. This is similar to how the human brain contains models of the tasks it can perform. But instead of brain power, AI models use a combination of algorithms, memory and processing power to solve their tasks.

The very fact that AI applications are driven by underlying models that can be used to reason about outcomes of certain actions or operations, renders these systems highly anticipatory. This means that many of these systems can be used in an anticipatory context, or that the systems themselves are able to anticipate the future, thereby changing their behavior. AI systems are often modeled in such a way that they take into account multiple possible (future) outcomes based on historical and present input. These possibilities are then used to reason about and decide on the next best course of action. For example, if a person has just listened to five classical piano pieces, he/she might be interested in more music of the same genre. But if that person has just left the office and went driving, we might know from past experience that that he/she likes a rougher genre while commuting home. However, it is 2pm, and usually that person only drives home after 5pm, which increases the uncertainty about whether we should recommend either classical or rock music. All the input that this AI system gets (musical taste, listening history, location, date and time, device...), can potentially be used in the final decision about what songs will be recommended to this person.

1.5 Machine learning

Artificial intelligence algorithms can be as simple as a set of rules that trigger a certain action. For example, a simple e-mail spam filter might check whether an incoming e-mail contains some predefined words or word groups, such as 'scam', '100% free', 'medicine', 'viagra', 'fast cash', 'free gift', dollar signs, exclamation marks, etc. If the e-mail contains a sufficient amount of these terms, it might be classified as spam. This is a fairly easy programming task for first-year bachelor students.

But if you tackle this problem in such a simple way, you might end up with a lot of false positives (genuine e-mails classified as spam) and false negatives (spam classified as genuine). First of all, what is 'a sufficient amount' of spam terms? Is it 5% of all words in the e-mail, is it less, is it more? This number or 'threshold' can be determined by a domain expert, but even then it is far from likely that this expert will pick the optimal threshold. Second, we assume that we have a set of predefined spam terms at our disposal. It is true that there exist such public lists, but they are far from complete, and since these lists are static and not changed often, true spammers can adapt their e-mail content to include as little of these known words as possible. So then, if we are unable to use a list of spam-triggering words, how would we identify them in a robust manner that can evolve over time? And finally, the contribution of every word to the final decision is ideally different. For example, the word 'free' is probably less characteristic of spam e-mails than 'fast cash', and if you practice a medial profession, you might actually receive a lot of valid e-mails containing the words 'medication' and 'drugs'. It is, however, an impossible task to determine manually to what extent a word is an indication of spam.

It is for all the above purposes that we use *machine learning*, as a special case of artificial intelligence. Kevin Murphy defines machine learning as "a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data" [23]. Let's discuss the elements in this definition. First of all, there's the data aspect of machine learning. We live in a time where, according to IBM®, we generate around 2.5 exabytes—i.e. 2.5 billion gigabytes—of data every day; this means that 90% of all data around has been generated in the last two years [24]. There is a wealth of information and knowledge present (or hidden) in all this data. Machine learning algorithms in general try to uncover this hidden information by processing large amounts of data, and to cast this information in a *model* that describes this data. This process is called learning or *training*, and is largely similar to how children learn to recognize objects: we give them a picture of a tree and say "this is a tree", and we will cor-

rect them if they claim that it is something else than a tree. But while the process of learning a good data model takes years for us humans, some machine learning algorithms can do it in a few hours or days, depending on the computational power at hand.

Next to data, Murphy states that machine learning algorithms learn patterns that are present in this data. A pattern can be anything that leads to conclusions about the data, and is the basic building block of a model. For example, if we present the algorithm many pictures of different objects, it might learn that a picture with two eyes in it probably shows some kind of animal. Or it might find from a dataset of rent prices that apartments in the city center are more expensive than in the outskirts. Ideally, machine learning algorithms find multiple such patterns in the presented data, so that they are able to build a robust model which can be used to make predictions and reason about future data. This depends of course highly, first of all, on the quality of the data that is presented. For example, if all apples in your dataset are green, then the algorithm might conclude that a red apple is not a real apple. And second, it also depends on the complexity of the model. Very simple models will ideally focus on large-scale patterns in the data, and only allow for broad and undetailed predictions. Too complex models, on the other hand, can learn very detailed patterns in the data, and, as a result, there exists an actual danger that the model will learn the complete dataset 'by heart'. This implies that, if your dataset contains three pictures of different apples, that the algorithm will have difficulties recognizing other apples. This phenomenon is called *overfitting*, and is one of the major challenges in machine learning. Effectively mitigating overfitting while maintaining a model that is of a high enough quality, is the goal of every machine learning application.

The definition of Murphy started by saying that machine learning is a 'set of methods'. In fact, there exist hundreds or even thousands of different machine learning models. There is no 'one model to rule them all', since each of the different models often have their own specialized purpose. The most high-level taxonomy divides machine learning models into either *supervised* learning, *unsupervised* learning or *reinforcement* learning models, which will be discussed briefly.

Supervised learning

In supervised learning, every entry x_i in the dataset has an associated label y_i. For example, x_i could be an e-mail message, and y_i indicates whether this e-mail is spam or not (1 or 0). Or x_i can be a full description of a city apartment, and y_i is the asked rent price. And a given picture x_j might

contain either a cat or a dog (1 or 0). The goal of supervised learning is to learn a model that is able to map a given x_i onto its label y_i as well as possible. That is, we learn a function $f\colon x_i \mapsto y_i$. Once we have learned this function f, we can apply it to new or unseen data x_j to predict its label $f(x_j) = \hat{y}_j$. We often use a hat symbol to indicate that a label is a prediction and not the real label or *ground-truth* label from the dataset. If the label is a real number or any quantity that can be ordered on an axis (an 'ordinal' variable), we use the term *regression*. If there are no orderings in the labels (e.g. between cats and dogs, or spam vs. no spam, i.e. a 'nominal' variable), we call it *classification*. In the context of sequences, a interesting supervised learning problem is to predict the next symbol in a given series. For example: predicting the next word or character in a text, predicting the next Apple® stock price every 5 seconds, predicting the next movie you will watch, etc. It is a supervised problem, since such machine learning algorithms are mostly trained on historical data, for which we know all 'future' symbols. The particular problem of sequence completion lies, among others, at the core of this doctoral thesis.

Unsupervised learning

If there are no labels in the dataset at hand, but we still want to uncover hidden patterns in the data, we call it unsupervised learning. The best-known family of methods in this group, are the *clustering* methods. These algorithms try to partition the dataset into groups of similar datapoints. For a dataset of cats and dogs images, a clever clustering algorithm might be able to separate these two classes just by looking at the images alone, but there are of course no guarantees. Another important class are the *dimensionality reduction* methods. Without going into too much (mathematical) details—those are reserved for the subsequent chapters—these methods try to explain the given data with fewer dimensions than present in the original data. For example, an image is built up of hundreds of pixels, but if we are just interested in separating cats from dogs, only one dimension might be enough to explain this difference. Another popular example that we have all come across, is the political compass[5]. In this compass, people are projected onto a two-dimensional plane according to two axes: left-right and libertarian-authoritarian. Figure 1.4 shows my own position in the political compass, along with several politicians. Such a representation can be seen as a dimensionality reduction: human beings are so complex that they can be described by thousands of different features, yet the compass projects all individuals along just two axes.

[5]https://www.politicalcompass.org

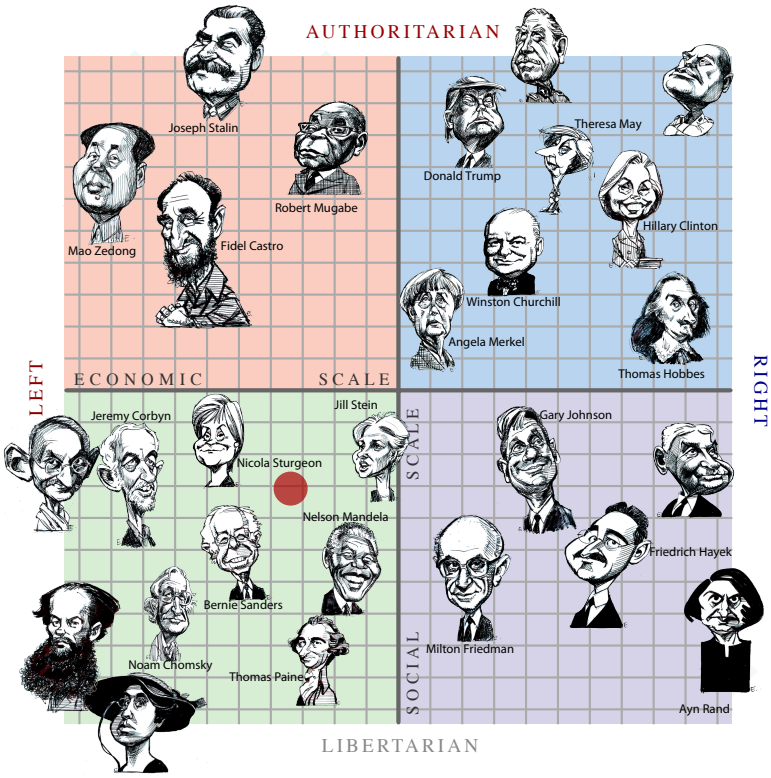


Figure 1.4: My personal political compass shows that I am slightly left and libertarian (as indicated by the red dot), close to Bernie Sanders and Nelson Mandela. Since the test is largely based on American society and economics, personally, I would have put myself much closer towards the center of the graph by European standards.

Reinforcement learning

A final class of machine learning models, is called reinforcement learning and can be compared to trial-and-error learning that occurs all throughout nature. Learning occurs by performing actions in the environment, and receiving reward or punishment based on the outcomes of the actions. Reinforcement learning is applied in the financial domain to automated stock trading[6], in the industrial domain to reduce the environmental footprint

[6]Noonan, Laura. "JPMorgan develops robot to execute trades", Financial Times, July 31st 2017.

of datacenters[7], in the medical domain to determine the optimal dosing scheme [25], and in the gaming domain to determine your most suited opponent on online multiplayer platforms[8]. Although many problems in supervised and unsupervised learning can be cast into a reinforcement learning problem—e.g. if you predict a cat, but it's a dog, you get punished—reinforcement learning is not the focus of this doctoral thesis.

Before we move to the next section, let's revisit the definition of machine learning by Murphy once again. He states that patterns are 'automatically' extracted from given data. Let's be critical about the word 'automatically' for a moment. For supervised learning, we have stated that the function f maps every x_i to y_i as 'best' as possible, and that, for unsupervised learning, clustering methods group 'similar' datapoints together. But these are rather vague statements. In the following, I will focus on supervised learning, although the argumentation is valid for other types of machine learning as well. First of all, as stated before, we have to counter overfitting: a function that predicts all labels perfectly might not be a good predictor for unseen data! But even more important, 'as well as possible' depends on how we define the so-called error or *loss* between the predicted label and the real label coming from the dataset. Suppose for example that the predicted rent price $\hat{y}_i$ for an apartment x_i is €470, while the real rent price y_i is €500. One might say that the error or difference between the two is €30, but that is only the case if you consider the so-called L_1 loss function[9], i.e. the absolute value of the difference between the two numbers:

$$\mathcal{L}_1(y_i, \hat{y}_i) = |y_i - \hat{y}_i| \,.$$

In machine learning, however, we often use the quadratic loss function—mainly because of theoretical grounds and optimization efficiency:

$$\mathcal{L}_2(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2.$$

In our example, the quadratic loss is equal to €^{2}900. If the predicted value moves further away from the real value, the quadratic loss will increase more heavily than the L_1 loss. Yet other loss functions might even attribute a greater punishment for underpriced than overpriced apartments. The choice of loss function will therefore heavily influence the behavior of the machine learning algorithm.

[7] Evens, Rich; Gao, Jim. "DeepMind AI reduces energy used for cooling Google data centers by 40%", Google Blog, July 20th 2016.

[8] Claim based on an industry talk by EA Games at RecSys, Como (Italy), 2017.

[9] Throughout this thesis, a loss function will generally be denoted by the calligraphic $\mathcal{L}$, although L, J, ... are also commonly used.

So, do machine learning algorithms learn patterns from data in an automated fashion? Yes and no. It is still up to the user or researcher to select the most appropriate model for the task, pick or tweak the loss function, select the data features that will be used, determine how the dataset will be structured and used for learning, etc. Once these choices and parameters are in place, then, yes, the machine learning algorithm will try to extract the necessary patterns according to the parameters you have set. We do not need to specify ourselves that larger apartments are usually more expensive than smaller ones; these conclusions should become apparent from the extracted patterns. If not, then you should go back to your initial premises: maybe the model is not powerful enough, maybe you have too little data, maybe the model is overfitting, etc. Methods such as cross-validation and meta-learning—dubbed 'learning to learn'—can help in this process, but even the best meta-model cannot solve all your machine learning problems[10].

1.6 Deep learning

A particular field of machine learning that has gained huge attention over the course of the last six or seven years, is *deep learning*. It is based on the concept of an (artificial) *neural network*, a machine learning model that—as the name suggests—mimics the firing of neurons in the (human) brain. Similar to how humans process signals coming from our eyes, ears, ... and attach a meaning to this input, neural networks can take images, sound, text, ... as input, feed it through the entire network, and produce an output that we can interpret. Although it is my intention to keep this introductory chapter as clear from technical details as possible, I cannot bypass some of core concepts of neural networks that will follow, so please bear with me for the next couple of paragraphs.

Figure 1.5 shows a toy example of a neural network. The basic building block of this neural network is a neuron that processes a signal. Neurons are indicated by small circles in the figure. The signals they process, are usually represented by real numbers. One neuron in the network takes an input signal and produces an altered version of this input as output

[10]This is one of the reasons I am skeptical about out-of-the-box machine learning solutions, also called Machine Learning as a Service (MLaaS). There is no doubt that AmazonML, Google Prediction and their pals can deliver very useful tools to anyone that wants to use machine learning in their applications. But these easy-to-use tools quickly lead to the perception that 'anyone can do machine learning', and it might even be dangerous if you have no understanding about the underlying mechanisms. It also leads to the rise of numerous AI, data insights and machine learning startups. Some of these company's employees often have little to no prior experience in the field, and a lot of larger firms are paying big money for their consulting sessions. Maybe there is an AI bubble on the verge of bursting?

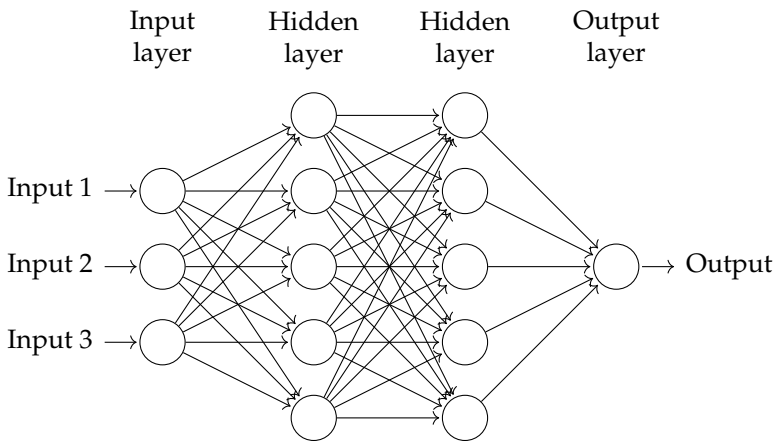


Figure 1.5: Toy example of a fully-connected neural network with four layers.

signal. This output signal is, subsequently, the input signal for another neuron, or it can be the final output of the network. Neurons are therefore interconnected and transfer information to and from each other, similar to the synapses in our brains, which is shown by the arrows in Figure 1.5.

To keep things as structured and manageable as possible, neurons are grouped into *layers*. The *layer size* or *dimensionality* refers to the number of neurons that are present in the layer. The example in the figure shows four layers: an input layer with size 3, two *hidden* layers of size 5—they are called hidden since their values are not observable at the input or output of the network—and one output layer with one neuron. The neural network is organized in such a way that all neuron outputs produced in one layer are processed by the neurons in another layer. Therefore, neurons that reside within the same layer do not communicate with each other.

The type of neural network shown in Figure 1.5 is called a *fully-connected* neural network, since the neurons in each layer are connected to all the neurons in the subsequent layer. A layer that is fully-connected to the next layer, is also called a *dense* or *fully-connected* layer. The network takes three values as input, and produces a single value by crossing four layers: the subsequent hidden layers transform the input in such a way that a suitable output value is calculated. In general, the power of neural networks is that the hidden layers are able to extract informative patterns from the input data in a hierarchical fashion. For example, if the input is an image of a cat, the first layers will detect edges in the picture, the next ones find rudimentary shapes, and the final hidden layers know how to look for ears, noses, eyes, nostrils, etc.

If we allow ourselves to be a tad formal, we refer back to the previous section on machine learning and state that a neural network computes one or more output values as a function of the input values. Every layer with N neurons is an ordered list of N real values, and the corresponding mathematical concept is a vector in the field $\mathbb{R}^N$. If the vector of input values is denoted by $\mathbf{x}$ and the output vector by $\mathbf{y}$, a neural network computes the function f:

$$\mathbf{y} = f(\mathbf{x}). \tag{1.5}$$

In the case of a fully-connected layer, the output of each neuron is multiplied by a real-valued *weight*, which essentially indicates the importance of that neuron's value in comparison to the other neurons in the same layer. Therefore, every arrow in Figure 1.5 has some associated weight w. If we look at the top neuron of the first hidden layer in this figure, we observe that a weighted signal from all three input neurons arrive at this hidden neuron. Before these signals are given to its input, they are summed together. Thus, the value of every neuron h_j^1 in the first hidden layer is a linear combination of all input values in $\mathbf{x}$:

$$\begin{aligned} h_1^1 &= w_{1,1} \cdot x_1 + w_{1,2} \cdot x_2 + w_{1,3} \cdot x_3, \\ h_2^1 &= w_{2,1} \cdot x_1 + w_{2,2} \cdot x_2 + w_{2,3} \cdot x_3, \\ &\dots \\ h_5^1 &= w_{5,1} \cdot x_1 + w_{5,2} \cdot x_2 + w_{5,3} \cdot x_3. \end{aligned} \tag{1.6}$$

In short, we write:

$$\mathbf{h}^1 = \mathbf{W}\mathbf{x}, \tag{1.7}$$

in which $\mathbf{W} \in \mathbb{R}^{5\times 3}$ is the matrix of the weights shown in Equation 1.6. The weights in these matrices will be learned or trained in order to calculate the desired output of the neural network.

We have now covered most elements of a basic neural network. However, there is still one ingredient missing. With what we know so far, the neural network shown in Figure 1.5 only calculates a linear transformation of the input, which is a restriction. The real power of neural networks is that they are able to learn non-linear transformations. They do this through the use of so-called *activation functions*. In neural networks, every output of a neuron is first put through such an activation function before it is sent to the next layer. Some widely used activation functions are ReLU ('rectified

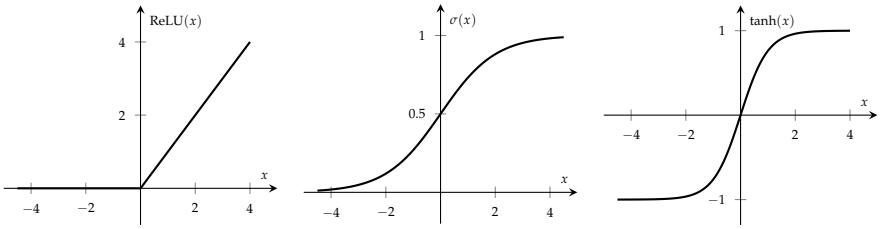


Figure 1.6: Graphical display of the ReLU, sigmoid and tanh activation functions.

linear unit'), sigmoid, tanh and softmax:

$$\text{ReLU}(x) = \max(x, 0), \tag{1.8}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{1.9}$$

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}, \tag{1.10}$$

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}, \ \forall i \in \{1 \dots K\}. \tag{1.11}$$

Figure 1.6 shows graphical plots for the ReLU, sigmoid and tanh functions. We observe that these activation functions limit or squash the values of the argument onto a certain interval: ReLU puts all negative values to zero and leaves the positive values untouched, while sigmoid and tanh confine the values to $[0, 1]$ and $[-1, 1]$, respectively. The softmax function is not shown; this particular activation function is applied to a vector of real values, and ensures that this vector becomes a valid, normalized probability mass function. This is useful if, for example, we have to decide whether an image contains a cat or a dog, a horse, a cow, ... In this case, the model gives us a probability estimate for each of the possibilities. Without going into further details, the choice of activation functions in the model should always be tuned to the problem at hand, which we will often discuss in the subsequent chapters. It should never be taken for granted.

With all the ingredients above, we can now look back at Figure 1.5 and write down the equations that calculate the output of the neural network based on the given inputs. Let's write an arbitrary activation function as ϕ, the inputs as $\mathbf{x}$, the outputs as $\mathbf{y}$ and the weight matrices as $\mathbf{W}^{ih}$, $\mathbf{W}^{hh}$ and $\mathbf{W}^{ho}$ for the input-to-hidden, hidden-to-hidden and hidden-to-output weights. We then get the following:

$$\mathbf{y} = \phi\Big(\mathbf{W}^{ho}\phi\Big(\mathbf{W}^{hh}\phi\Big(\mathbf{W}^{ih}\mathbf{x}\Big)\Big)\Big). \tag{1.12}$$

What we observe is an equation of nested matrix multiplications and non-

linear transformations. In practice, we also add a vector with learned parameters $\mathbf{b}$—also called a *bias*—to each matrix multiplication:

$$\mathbf{y} = \phi\Big(\mathbf{W}^{ho}\phi\Big(\mathbf{W}^{hh}\phi\Big(\mathbf{W}^{ih}\mathbf{x} + \mathbf{b}^{ih}\Big) + \mathbf{b}^{hh}\Big) + \mathbf{b}^{ho}\Big)\,. \tag{1.13}$$

If we have knowledge of all weight matrices, biases and activation functions, then we see that, indeed, the neural network is a computable function of the input $\mathbf{x}$. By careful selection of the weights, a neural network is able to model very complex functions, and it has been shown that (feed-forward) neural networks are 'universal function approximators' [26].

As we have discussed previously for general machine learning techniques, training a neural network—that is, choosing the optimal weights and biases—is a data-driven process: we apply some data at the input of the network, we observe the output, and we compare this predicted output to what the real output should be using a predefined loss function. The weights themselves are then learned through a process called *backpropagation*. Without going into too many technical details, backpropagation is essentially a clever application of the chain rule for computing the derivative or gradient of the loss function with respect to all parameters in the network. This gradient is indicative of how and to what extent we should alter the weights in the network. Figure 1.7 shows a toy example of a loss curve for two weights. The gradient in point C is shown in green; by following the arrow in the opposite direction, we get closer to a region in which the loss is lower. Points A and B are called *local minima*, while point B is the *global minimum*, i.e. the best weight configuration.

As we have shown in Equation (1.13), a neural network function is a nested calculation of matrix multiplications and non-linearities. If the network becomes deeper by adding more layers, or if the layer size increases heavily—i.e. we change the model's *architecture*—we allow the network to learn more complex functions. However, there is a associated computational cost, both to evaluate the neural network function, but also, and most importantly, the training process slows down significantly. This is one of the reasons[11] why neural networks became out of fashion by the nineties and noughties, and were overtaken in the AI community by so-called Support Vector Machines (SVMs). However, the advent of more computing power, mostly in the form of Graphics Processing Units (GPUs)[12], neural networks have gained huge popularity again since the beginning of

[11] Also, already in the 1960's Minsky and Papert showed that simple one-layer neural networks ('perceptrons') are unable to solve the XOR problem [27].

[12] GPUs are designed to process images as efficiently as possible. Since images are essentially tables of pixel values, they are represented by matrices. And because neural networks rely heavily on matrix calculations, GPUs are the preferred hardware for the task.

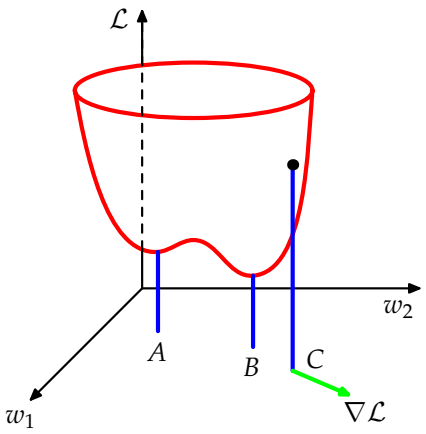


Figure 1.7: Example of a loss function for two weights w_1 and w_2. The gradient of the loss in point C is indicated by the green arrow. Points A and B show two local minima, and point B is the global minimum. Adapted after [26].

this decade. This evolution also resulted in the development and research of networks with many and large layers. Essentially, this is how the term 'deep learning' came into use, and it means nothing more than 'machine learning using neural networks with many layers'.

Traditionally, the begin date of the deep learning hype is marked 2012, when Alex Krizhevsky won the international ImageNet competition by a large margin using a deep neural network [28]. He won the competition using a so-called *convolutional neural network*, which borrows its structure from insights in the field of image processing. Although immensely popular and powerful, convolutional neural networks are not the focus of this thesis. Another widely-used class of neural networks, are the *recurrent neural networks* (RNNs). These networks allow that the output of a particular layer is reused as input for the same layer one of the previous layers. If we recall Equation (1.3), we can imagine that such recurrent connections in the network are very useful to model sequences. That is, by activating the recurrent connections for every new symbol in the sequence, a value produced at position n in the sequence can be used to calculate the value at position $n+1$. RNNs are therefore the models of choice in anticipatory applications, since they can be used to predict future values in a sequence. Since this doctoral thesis focuses heavily on learning with and about sequential data, one can imagine that RNNs will play an important role in the subsequent chapters. In the remainder of this introductory chapter, I will not go further into the details of RNN modeling and training. For this, I refer the reader to Chapter 4.

1.7 Representation learning

The real power of deep models is that they are able to take raw data as their input. By raw data we mean the actual pixel values in an image or video frame, all the ordered words in a text, a complete sound wave, etc. By contrast, most traditional machine learning methods require a preprocessed data format. Finding efficient data representations is a research field in itself called *feature engineering*. In the context of the political compass, if we would take the left-right axis as one feature, and the authoritarian-libertarian axis as second feature, I am roughly represented by the vector $[-0.33, -0.31]$; Donald Trump, on the other hand, is given by $[0.3, 0.8]$. Deep neural networks mainly don't require such feature engineering, but are able to learn useful and important features from the data itself. They can be extracted from one or more of the hidden layers in the network.

Suppose for example that we have a neural network that has learned to classify images. Once the network is trained, we can apply an image to its input, and observe the neuron values in all layers. Then, if one of the (hidden) layers has a size of 128 neurons, the image is represented by a 128-dimensional vector at that position in the network, i.e. the image is projected into a 128-dimensional space. Such a vector is called a representation, a *feature vector* or an *embedding* of the image. The greatest benefit is that we do not have to define what the 128 dimensions mean. The downside is that the dimensions become hard to interpret. For example, if we have a neural network that takes all information from a person as input and compresses it in a two-dimensional representation, we do not know immediately what the presented information means, as opposed to the two axes in the political compass.

Traditionally, the final layers in a neural network are considered the most useful representations of the data, depending of course on the application at hand. In the context of sequences and recurrent neural networks, the representations can even change with every new symbol that is processed. The representations of the sequence are therefore dynamic reflections of how the sequence evolves and what the sequence will look like in the future.

Deep learning is often associated with the concept of *representation learning*; to many, these terms are even synonymous. To me—but this is of course a personal point of view—the learning of representations is often a nice benefit or side-effect of using deep learning models. But deep neural networks can also be tailored explicitly to this purpose, and then it becomes highly associated with the concept of unsupervised learning and dimensionality reduction. It is about projecting objects in a space with few

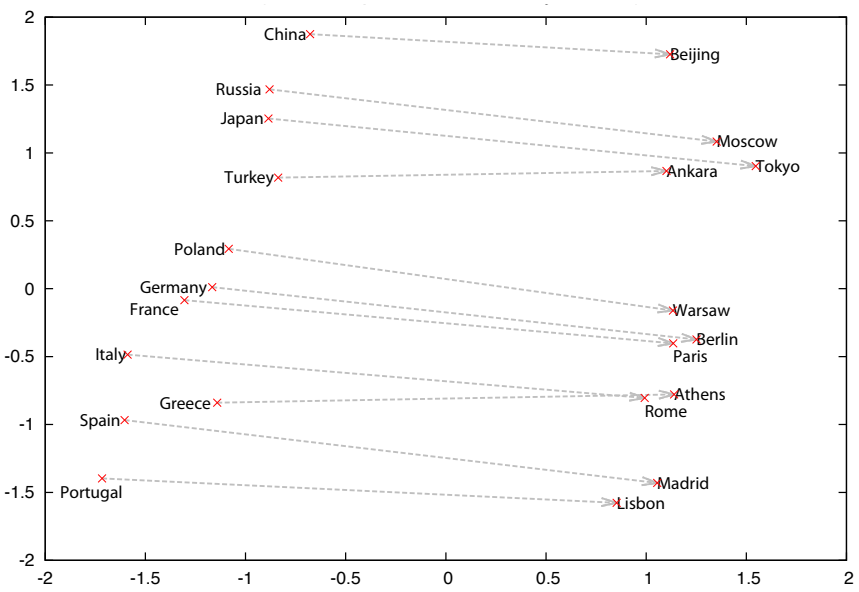


Figure 1.8: A 2-dimensional reduction of 1000-dimensional word embeddings for several countries and their corresponding capitals. Copied from (and more details to be found in) [30].

dimensions such that we can observe or extract useful characteristics of the objects in this space. For example, if two objects are positioned close to each other, they are in a sense related, or they share some common characteristics. In this context, recall the political compass from Figure 1.4.

A clear and nice example of representation learning, is the word2vec algorithm invented by Tomas Mikolov in 2013 [29, 30], which will play a central role in the following chapters. Word2vec is essentially a simple neural network that creates representations for words appearing in a large text corpus, e.g. Wikipedia, news articles, Trump tweets, lyrics, etc. It does this by taking a word representation in a sentence and by predicting the words that appear in its neighborhood. For example, in the opening quote of this chapter, word2vec will try to predict 'fascination', 'exaltation' and 'revelation' based on the word representation of 'temptations'. After processing thousands or millions of these quotes, the representations appearing in the final layer of the neural network will (hopefully) have reached stable values. We note that word representations are best known as *word embeddings* in the research community.

After training, the word2vec space starts to show some interesting characteristics. Figure 1.8 shows the word embeddings for several countries

Table 1.1: For a given word, the ten closest words in word2vec space are given in descending order, based on cosine distance with 400-dimensional word embeddings trained on the English Wikipedia dump from March 2015.

water	**chips**	**ghent**	**violin**
seawater	chip	antwerp	cello
groundwater	velveeta	bruges	piano
potable	omelets	brussels	clarinet
drinkable	breadsticks	maastricht	harpsichord
tubewells	edram	kortrijk	violins
greywater	twisties	oudenaarde	bassoon
desalinated	wafers	utrecht	violoncello
outfalls	microwavable	mechelen	flute
borewells	fritos	tournai	viola
nonpotable	eprom	vooruit	oboe

associated with their capitals. The original embeddings had 1000 dimensions[13], but were projected onto a 2D plane using Principal Components Analysis (PCA) [30]. We immediately observe a clear structure in the word embedding space: all countries appear on the left, while the capitals are situated on the right. On top of that, going from a country to its corresponding capital, we always have to travel more or less in the same direction. It even turns out that, at least in the original embedding space, we can do 'word embedding arithmetic', such as:

$$\mathbf{v}_{\text{Berlin}} \approx \mathbf{v}_{\text{Paris}} - \mathbf{v}_{\text{France}} + \mathbf{v}_{\text{Germany}}. \tag{1.14}$$

That is, subtracting the word embedding of France from the Paris embedding, and adding the embedding of Germany, brings us (very) close to the word embedding of Berlin.

Another useful property of word2vec spaces, is that word *similarity* can easily be measured. Similar words words appear frequently together in the text corpus. This is expressed in the word2vec space by the distance between the word embeddings. The closer two words appear in the space, the more they share one or more common meanings. An example will probably make the idea more tangible. In Table 1.1 I have listed four different words, and for each word the ten closest words are shown. The word embedding of 'ghent' lies close to other cities in the Benelux, and also 'vooruit'—a famous historical building and cultural venue in Ghent—appears in the top 10. For 'violin' we are given other popular classical instruments. If we take look at 'water', the closest words are related substantives and adjectives that are frequently used in the same context (potable,

[13]Note that 1000 is relatively small compared to the size of a typical word vocabulary.

desalinated...). I have included 'chips' as well, since it shows words that are related to the concept of potato chips (breadsticks, twisties, fritos), to electronic chips (edram, eprom), or to both (wafers). This example shows that word embeddings are able to capture multiple *semantic* meanings, given enough dimensions.

So, why would you need high-quality and low-dimensional data representations, such as word embeddings? By far the best use case for such embeddings, is to allow for fast search and retrieval of similar objects. Such systems are nowadays used in state-of-the-art recommender systems, which we have discussed in the taxonomy of Section 1.4. An example of how such a system is used within Spotify will be discussed in Chapter 6. Another use case was illustrated by Equation (1.14). Although the example is not useful in itself, in general, addition and subtraction of word embeddings are mathematical operations that produce sensible results. This means that if we sum all embeddings from the words that appear in the same sentence, we calculate a meaningful *sentence embedding*. Summing all the embeddings from books that one person has read, leads to a person embedding or *user embedding* that summarizes that person's reading interests. This user profile can then be used to make predictions about, for example, a user's future reading behavior. This very idea of manipulating embeddings, and especially embeddings that summarize sequences, is the major cornerstone of this thesis.

1.8 Research contributions

This doctoral dissertation is comprised of five research-focused chapters. Each chapter tackles a particular research question in the context of learning informative and high-quality representations for sequential data. For this purpose, most questions will be approached in the context of several applications, which I will briefly discuss below.

Chapter 2 – Detecting events on Twitter

The first chapter tackles the problem of extracting event information from a stream of Twitter messages, also called tweets. Examples of detected events are football games, tv shows, band concerts, etc. In this application, we encounter sequential data on multiple levels. First, can we find a useful representation for each tweet as a sequence of words, for which we will use traditional feature engineering? Second, how do we process the incoming stream of tweets in order to determine clusters of events?

Chapter 3 – Representing very short texts

In the subsequent chapter we build upon the insights of the previous chapter. Traditional text features turn out not to be informative in the context of Twitter, in one respect due to noise and word corruption, and in the other because of the low character count—Twitter used to allow only 140 characters per tweet. How then do we come up with high-quality representations for very short texts, such as tweets and text messages?

Chapter 4 – Training recurrent neural networks

This chapter goes deep into the details of recurrent neural networks. By investigating different training and sampling procedures, do we observe a performance shift of the network? Four alternatives are studied to train and make predictions from a recurrent neural network, and we give some go-to advice for any researcher interested in this area of deep learning.

Chapter 5 – Composing piano music

Using the insights from the previous chapter, we use recurrent neural networks to find dynamic representations for classical piano pieces. Are these networks able to model and generate new music? And can we tune the composition process to the style of a certain composer, such as Bach and Beethoven?

Chapter 6 – Modeling Spotify users

In the final research chapter, we show how recurrent neural networks can be used to model users in the large-scale recommender systems of Spotify. These user models are reflections of a user's music taste, and are used to recommend new songs. Are recommender systems based on RNNs able to beat non-sequential state-of-the-art recommenders?

1.9 Publications

The research results obtained during this PhD research have been published in scientific journals and presented at a series of international conferences and workshops. The following list provides an overview of these publications.

1.9.1 Publications in international journals (listed in the Science Citation Index[14])

[1] **C. De Boom**, S. Van Canneyt, T. Demeester, and B. Dhoedt, *Representation Learning for Very Short Texts using Weighted Word Embedding Aggregation*. Published in Pattern Recognition Letters, 80:150–156, 2016.

[2] **C. De Boom**, R. Agrawal, S. Hansen, E. Kumar, R. Yon, C.-W. Chen, T. Demeester, and B. Dhoedt, *Large-scale User Modeling with Recurrent Neural Networks for Music Discovery on Multiple Time Scales*. Published in Multimedia Tools and Applications, online, 2017.

[3] **C. De Boom**, M. De Coster, D. Spitael, S. Leroux, S. Bohez, T. Demeester, and B. Dhoedt, *Polyphonic Piano Music Composition with Composer Style Injection using Recurrent Neural Networks*. Submitted to Neural Computing and Applications, December, 2017.

[4] **C. De Boom**, T. Demeester, and B. Dhoedt, *Character-level Recurrent Neural Networks in Practice: Comparing Training and Sampling Schemes*. Published in Neural Computing and Applications, online, 2018.

1.9.2 Publications in international conferences (listed in the Science Citation Index[15])

[1] **C. De Boom**, J. De Bock, A. Van Camp, and G. de Cooman, *Robustifying the Viterbi Algorithm*. Published in Lecture Notes in Artificial Intelligence, 8754:160–175, 2014.

[14]The publications listed are recognized as 'A1 publications', according to the following definition used by Ghent University: A1 publications are articles listed in the Science Citation Index, the Social Science Citation Index or the Arts and Humanities Citation Index of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper.

[15]The publications listed are recognized as 'P1 publications', according to the following definition used by Ghent University: P1 publications are proceedings listed in the Conference Proceedings Citation Index - Science or Conference Proceedings Citation Index - Social Science and Humanities of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper, except for publications that are classified as A1.

[2] **C. De Boom**, S. Van Canneyt, S. Bohez, T. Demeester, and B. Dhoedt, *Learning Semantic Similarity for Very Short Texts*. Published in IEEE International Conference on Data Mining Workshop (ICDMW), Atlantic City (NJ), USA, 2015.

[3] R. Lemahieu, S. Van Canneyt, **C. De Boom**, and B. Dhoedt, *Optimizing the Popularity of Twitter Messages through User Categories*. Published in IEEE International Conference on Data Mining Workshop (ICDMW), Atlantic City (NJ), USA, 2015.

[4] J. van der Hooft, **C. De Boom**, S. Petrangeli, T. Wauters, F. De Turck, *AHTTP/2 Push-Based Framework for Low-Latency Adaptive Streaming Through User Profiling*. Published in IEEE/IFIP Network Operations and Management Symposium (NOMS), Taipei, Taiwan, 2018.

1.9.3 Publications in other international conferences

[1] **C. De Boom**, S. Van Canneyt, and B. Dhoedt, *Semantics-driven Event Clustering in Twitter Feeds*. Published in Proceedings of the 5th Workshop on Making Sense of Microposts (#Microposts), Florence, Italy, 2015.
This paper received the Best Paper Award.

[2] R. Houthooft, **C. De Boom**, S. Verstichel, F. Ongenae, and F. De Turck, *Structured Output Prediction for Semantic Perception in Autonomous Vehicles*. Published in 30th AAAI Conference on Artificial Intelligence, Phoenix (AZ), USA, 2016.

[3] **C. De Boom**, S. Leroux, S. Bohez, P. Simoens, T. Demeester, and B. Dhoedt, *Efficiency Evaluation of Character-level RNN Training Schedules*. Published in the ICML Data Efficient Machine Learning workshop, New York (NY), USA, 2016.

[4] S. Leroux, S. Bohez, **C. De Boom**, E. De Coninck, T. Verbelen, B. Vankeirsbilck, P. Simoens, and B. Dhoedt, *Lazy Evaluation of Convolutional Filters*. Published in the ICML Workshop on Device Intelligence, New York (NY), USA, 2016.

1.9.4 Poster publications in international conferences

[1] **C. De Boom**, S. Van Canneyt, T. Demeester, and B. Dhoedt, *Learning Representations for Tweets Through Word Embeddings*. Published in Proceedings of the Belgian-Dutch Conference on Machine Learning (BeneLearn), Kortrijk, Belgium, 2016.

References

[1] S. Pinker. *The Language Instinct*. How the Mind Creates Language. Penguin UK, February 2003.

[2] N. Chomsky. *Tool Module: Chomsky's Universal Grammar* [online]. Available from: http://thebrain.mcgill.ca/flash/capsules/outil_rouge06.html.

[3] G. Rawlinson. *The significance of letter position in word recognition*. Ieee Aerospace and Electronic Systems Magazine, 22(1):26–27, January 2007.

[4] C. E. Shannon. *Communication in the Presence of Noise*. Proceedings of the IRE, 37(1):10–21, 1949.

[5] N. Rescher. *Process Metaphysics*. An Introduction to Process Philosophy. SUNY Press, 1996.

[6] R. Poli. *Introduction to Anticipation Studies*. Springer, August 2017.

[7] B. Adam. *Timescapes of modernity: The environment and invisible hazards*, 1998.

[8] A. Falcon. *Aristotle on causality* [online]. 2006. Available from: https://plato.stanford.edu/archives/spr2015/entries/aristotle-causality/.

[9] J. Beckert. *Capitalism as a System of Fictional Expectations*. Politics & Society, 41(3):323–350, 2013.

[10] T. Suddendorf, D. R. Addis, and M. C. Corballis. *Mental time travel and the shaping of the human mind*. Philosophical Transactions of the Royal Society of London B: Biological Sciences, 364(1521):1317–1324, May 2009.

[11] M. S. Schulz. *Future moves: Forward-oriented studies of culture, society, and technology*. Current Sociology, 63(2):129–139, 2015.

[12] R. Rosen. *Anticipatory Systems*. Philosophical, Mathematical, and Methodological Foundations. Springer, New York, 2nd edition, 2012.

[13] D. B. Huron. *Sweet Anticipation*. Music and the Psychology of Expectation. MIT Press, 2006.

[14] S. Russell and P. Norvig. *Artificial Intelligence: Pearson New International Edition*. A Modern Approach. Pearson Higher Ed, August 2013.

[15] A. Wang. *An Industrial Strength Audio Search Algorithm*. In ISMIR, 2003.

[16] F.-h. Hsu. *Behind Deep Blue*. Building the Computer that Defeated the World Chess Champion. Princeton University Press, February 2004.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. *Human-level control through deep reinforcement learning.* Nature, 518(7540):529–533, 2015.

[18] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. *Mastering the game of Go without human knowledge*. Nature, 550(7676):354–+, 2017.

[19] N. Jaques, S. Gu, R. E. Turner, and D. Eck. *Generating Music by Fine-Tuning Recurrent Neural Networks with Reinforcement Learning*. In Deep Reinforcement Learning Workshop, NIPS, December 2016.

[20] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. *WaveNet: A Generative Model for Raw Audio*. arXiv.org, September 2016. arXiv:1609.03499v2.

[21] L. A. Gatys, A. S. Ecker, and M. Bethge. *A Neural Algorithm of Artistic Style*. arXiv.org, August 2015. arXiv:1508.06576v2.

[22] B. Baker, O. Gupta, N. Naik, and R. Raskar. *Designing Neural Network Architectures using Reinforcement Learning.* CoRR, cs.LG, 2016.

[23] K. P. Murphy. *Machine Learning*. A Probabilistic Perspective. MIT Press, August 2012.

[24] M. C. IBM. *10 Key Marketing Trends for 2017* . public.dhe.ibm.com, 2017.

[25] S. Nemati, M. M. Ghassemi, and G. D. Clifford. *Optimal medication dosing from suboptimal clinical examples - A deep reinforcement learning approach.* EMBC, pages 2978–2981, 2016.

[26] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer Verlag, August 2006.

[27] M. Minksy and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969.

[28] A. Krizhevsky, I. Sutskever, and G. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. In NIPS 2012, November 2012.

[29] T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. In Proceedings of Workshop at ICLR, January 2013.

[30] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. *Distributed Representations of Words and Phrases and their Compositionality*. In NIPS 2013: Advances in neural information processing systems, October 2013.

2

Semantics-driven Event Clustering in Twitter Feeds

"Every day they shout and scold and go about their lives, heedless of the gift it is to be them."

—Quasimodo, 1996

Twitter is one of the most popular social media platforms where you post short messages that are publicly visible. The maximum number of characters per 'tweet' used to be 140, but as of November 7 2017, it has been doubled to 280 characters. According to Internet Live Stats[1], around 8,000 tweets are sent every second. By tapping into this huge information stream, we can potentially extract useful knowledge. This chapter focuses on identifying events using the Twitter stream in an automated fashion. That is, we want to detect whether a football game, music concert, television show... is currently taking place based on tweets sent by different users. In this, we will face two sequential information challenges. First, there is the problem of adequately representing a tweet based on the words and information it contains. Second, since tweets present themselves one after another in such a high volume, our algorithms should be able to cope with sequential information extraction.

⋆⋆⋆

[1] www.internetlivestats.com, measured on February 5 2018, 1pm.

C. Boom, S. Van Canneyt, and B. Dhoedt.

Appeared in proceedings of the 5th Workshop on Making Sense of Microposts. 1395. p.2-9

Abstract Detecting events using social media such as Twitter has many useful applications in real-life situations. Many algorithms which all use different information sources—either textual, temporal, geographic, community... knowledge—have been developed to achieve this task. Semantic information is often added at the end of the event detection to classify events into semantic topics. But semantic information can also be used to drive the actual event detection, which is less covered by academic research. We therefore supplemented an existing baseline event clustering algorithm with semantic information about the tweets in order to improve its performance. This paper lays out the details of the semantics-driven event clustering algorithms developed, discusses a novel method to aid in the creation of a ground truth for event detection purposes, and analyses how well the algorithms improve over baseline. We find that assigning semantic information to every individual tweet results in just a worse performance in F_1 measure compared to baseline. If however semantics are assigned on a coarser, hashtag level the improvement over baseline is substantial and significant in both precision and recall. Semantic information of tweets can thus indeed be used to improve the performance of existing and new event detection algorithms.

2.1 Introduction

Traditional media mainly cover large, general events and thereby aim at a vast audience. Events that are only interesting for a minority of people are rarely reported. Next to the traditional mass media, social media such as Twitter and Facebook are a popular source of information as well, but extracting valuable and structured data from these media can be challenging. Posts on Twitter for example have a rather noisy character: written text is mostly in colloquial speech full of spelling errors and creative language use, such posts often reflect personal opinions rather than giving an objective view of the facts, and a single tweet is too short to grasp all the properties that represent an event. Nevertheless the user-contributed content on social media is extensive, and leveraging this content to detect events can complement the news coverage by traditional media, address more selective or local audiences and improve the results of search engines.

In the past researchers mostly used textual features as their main source

of information to perform event detection tasks in social media posts. Next to the text itself, other characteristic features such as the timestamp of the post, user behavioural patterns and geolocation have been successfully taken into account [1–6]. Less used are so-called semantic features, in which higher-level categories or semantic topics are captured for every tweet and used as input for the clustering algorithm. These semantic topics can either be very specific—such as sports, politics, disasters. . . —or can be latent abstract categories not known beforehand; such an abstract topic is usually a collection of semantically related words. In most applications semantics are determined on event level after the actual event detection process [7]. We however propose to use semantic information on tweet level to drive the event detection algorithm. After all, events belonging to different semantic categories—and thus also its associated tweets—are likely to be discerned more easily than semantically related events. For example, it is relatively easy to distinguish the tweets of a sports game and a concurrent politics debate.

The use case we address in this paper consists of dividing a collection of tweets into separate events. In this collection every tweet belongs to a certain event and it is our task to cluster all tweets in such a way that the underlying event structure is reflected through these clusters of tweets. For this purpose we adopt a single pass clustering mechanism. As a baseline we use a clustering approach which closely resembles the algorithm proposed by Becker et al. to cluster Flickr photo collections into events [8, 9], and in which we only use plain textual features. We then augment this baseline algorithm, now incorporating semantic information about the tweets as a second feature next to the text of the tweet. As it turns out, solely using a semantic topic per tweet only marginally improves baseline performance; the attribution of semantic labels on tweet level seems to be too fine-grained to be of any predictive value. We therefore employ an online dynamic algorithm to assign semantic topics on hashtag level instead of tweet level, which results in a coarser attribution of topic labels. As will be shown in this paper, the latter approach turns out to be significantly better than baseline performance.

The remainder of this paper is structured as follows. In Section 2.2 we shortly discuss the most appropriate related work in recent literature, after which we describe the methodology to extract events from a collection of Twitter posts in Section 2.3. The collection of data and the construction of a ground truth is treated in Section 2.4. Finally we analyse the results of the developed algorithms in Section 2.5.

2.2 Related work

Since the emergence of large-scale social networks such as Twitter and their growing user base, the detection of events using social information has attracted the attention of the scientific community. In a first category of techniques, Twitter posts are clustered using similarity measures. These can be either based on textual, temporal, geographical or other features. Becker et al. were among the first to implement this idea by clustering a Flickr photo collection [8, 9]. They developed a single pass unsupervised clustering mechanism in which every cluster represented a single event. Their approach however scaled exponentially in the number of detected events, leading to Reuter et al. improving their algorithm by using a prior candidate retrieval step [3], thereby reducing the execution time to linear scaling. Petrović et al. used a different technique based on Locality Sensitive Hashing, which can also be seen as a clustering mechanism [10]. In this work, tweets are clustered into buckets by means of a hashing function. Related tweets are more probable to fall into the same bucket, which allows for a rapid comparison between tweets to drive the event detection process.

The techniques in a second category of event detection algorithms mainly use temporal and volumetric information about the tweets being sent. Yin et al. for example use a peak detection strategy in the volume of tweets to detect fire outbreaks [6], and Nichols et al. detect volume spikes to identify events in sporting games [11]. By analysing communication patterns between Twitter users, such as peaks in original tweets, retweets and replies, Chierichetti et al. were able to extract the major events from a World Cup football game or the Academy Awards ceremony [12]. Sakaki et al. regarded tweets as individual sensor points to detect earthquakes in Japan [4]. They used a temporal model to detect spikes in tweet volume to identify individual events, after which a spatial tracking model, such as a Kalman filter or a particle filter, was applied to follow the earthquake events as they advanced through the country. Bursts of words in time or in geographic location can also be calculated by using signal processing techniques, e.g. a wavelet transformation. Such a technique was successfully used by Weng et al. in their EDCoW algorithm to detect Twitter events [13], and by Chen and Roy to detect events in Flickr photo collections on a geographic scale [14].

Semantic information is often extracted after the events are detected to classify them into high level categories [15]. This can be done in either a supervised way, using a classifier like Naive Bayes or a Support Vector Machine, but most of the times unsupervised methods are preferred, since they do not require labelled data to train models and are able to discover

semantic categories without having to specify these categories beforehand. Popular unsupervised techniques are Latent Dirichlet Allocation (LDA), clustering, Principal Component Analysis (PCA) or a neural auto-encoder. LDA was introduced by Blei et al. in 2003 as a generative model to extract latent topics from a large collection of documents [16]. Since then many variants of LDA have emerged tailored to specific contexts. Zhao et al. created the TwitterLDA algorithm to extract topics from microposts, such as tweets, assuming a tweet can only have one topic. Using community information next to purely textual information, Liu et al. developed their own version of LDA as well, called Topic-LinkLDA [17]. A temporal version of LDA, called TM-LDA, was developed by Wang et al. to be able to extract topics from text streams, such as a Twitter feed [18]. By batch grouping tweets in hashtag pools, Mehrotra et al. were able to improve standard LDA topic assignments to individual tweets [19].

2.3 Event clustering

In this section we will describe the mechanics to discover events in a collection of tweets. In the dataset we use, every tweet t is assigned a set of event labels E_t. This set contains more than one event label if the tweet belongs to multiple events. The dataset itself consists of a training set T_{train} and a test set T_{test}. The details on the construction of the dataset are found in Section 2.4. We will now try to recover the events in the test set by adopting a clustering approach. First the mechanisms of an existing baseline algorithm will be expounded. Next we will extend this algorithm using semantic information calculated from the tweets.

2.3.1 Baseline: single pass clustering

Our baseline algorithm will use single pass clustering to extract events from the dataset. Becker et al. elaborated such an algorithm to identify events in Flickr photo collections [8, 9]; their approach was criticized and improved by Reuter et al. for the algorithm to function on larger datasets [3]. In this paper we will adopt single-pass clustering as a baseline that closely resembles the algorithm used by Becker et al.

As a preprocessing step, every tweet in the dataset is represented by a plain tf-idf vector[2] and sorted based on its timestamp value. In the following we will use the same symbol t for the tweet itself and for its tf-idf vector. As the algorithm proceeds, it will create clusters of tweets, which are the retrieved events. We denote the cluster to which tweet t belongs as

[2]We refer to Section 3.2 for a clear-cut definition of tf-idf.

S_t; this cluster is also characterized by a cluster center point s_t. We refer to a general cluster and corresponding cluster center point as resp. S and s. The set A contains all clusters which are currently active, i.e. being considered in the clustering procedure. During execution of the algorithm, a cluster is added to A if it is newly created. After some time a cluster can become inactive by removing this cluster from the set A. In Section 2.5 we will specify how a cluster can become inactive.

The baseline algorithm works as follows. When the current tweet t is processed, the cosine similarity $\cos(t, s)$ between t and cluster center s is calculated for all S in A. A candidate cluster S'_t (with cluster center s'_t) to which t could be added, and the corresponding cosine similarity $\cos(t, s'_t)$, are then calculated as

$$S'_t = \arg\max_{S \in A} \cos(t, s), \tag{2.1}$$

$$\cos(t, s'_t) = \max_{S \in A} \cos(t, s). \tag{2.2}$$

If S'_t does not exist—this occurs when A is empty—we assign t to a new empty cluster S_t, we set $s_t = t$ and S_t is added to A. If S'_t does exist, we need to decide whether t belongs to this candidate cluster or not. For this purpose we train a logistic regression classifier from LIBLINEAR [20] with a binary output. It takes $\cos(s'_t, t)$ as a single feature and decides whether t belongs to S'_t. If it does, then we set S_t to S'_t and we update its cluster center s_t as follows:

$$s_t = \frac{\sum_{t \in S_t} t}{|S_t|}. \tag{2.3}$$

If t does not belong to S'_t according to the classifier, then as before we assign t to a new empty cluster S_t and we set $s_t = t$.

In the train routine we calculate the candidate event cluster S'_t for every tweet t in T_{train} and verify whether this cluster corresponds to one of the event labels of t in the ground truth. If it does, we have a positive train example, otherwise a negative example. The number of positive and negative examples are balanced by randomly removing examples from either the positive or negative set, after which the examples are used to train the classifier.

In the original implementation by Becker et al. the processing of a tweet is far from efficient since every event cluster has to be tested. After a certain time period, the amount of clusters becomes very large. The adjustments by Reuter et al. chiefly aim at improving this efficiency issue. We do not consider these improvements here, since in Equation (2.1) we only test currently active clusters, which is already a performance gain.

2.3.2 Semantics-driven clustering

To improve the baseline single pass clustering algorithm we propose a clustering algorithm driven by the semantics of the tweets. For example tweets that belong to the same semantic topic—e.g. sports, disasters, ...—are more likely to belong to the same event than tweets about different topics. Discerning two events can become easier as well if the two events belong to different categories.

To calculate a semantic topic for each of the tweets in the dataset, we make use of the TwitterLDA algorithm [21]. It is an adjustment of the original LDA (Latent Dirichlet Allocation) algorithm [16] for short documents such as tweets, in which every tweet only gets assigned a single topic—instead of a probabilistic distribution over all the topics—and single user topic models are taken into account. After running the TwitterLDA algorithm, every tweet t gets assigned a semantic topic γ_t.

The actual clustering algorithm has the same structure as the baseline algorithm, but it uses the semantic topic of the tweets as an extra semantic feature during clustering. We define the semantic fraction $\sigma(t, S)$ between a tweet and an event cluster as the fraction of tweets in S that have the same semantic topic as t:

$$\sigma(t, S) = \frac{|\{t' : t' \in S \wedge \gamma_{t'} = \gamma_t\}|}{|S|}. \tag{2.4}$$

To select a candidate cluster S'_t (with cluster center s'_t) to which t can be added, we use the cosine similarity, as before, as well as this semantic fraction:

$$S'_t = \arg\max_{S \in A} \cos(t, s) \cdot \sigma(t, S). \tag{2.5}$$

We choose to multiply cosine similarity and semantic fraction to select a candidate cluster since both have to be as large as possible, and if one of the two factors provides serious evidence against the candidate cluster, we want this to be reflected. Now we use both $\cos(t, s'_t)$ and $\sigma(t, S'_t)$ features to train a logistic regression classifier with a binary output. The rest of the algorithm continues in the way the baseline algorithm does.

2.3.3 Hashtag-level semantics

As pointed out by Mehrotra et al. the quality of topic models on Twitter data can be improved by assigning topics to tweets on hashtag level instead of on tweet level [19]. To further improve the semantics-driven clustering, we therefore use a semantic majority voting scheme on hashtag

level, which differs from the approach by Mehrotra et al. in that it can be used in an online fashion and that we consider multiple semantic topics per tweet.

In the training set we assign the same topic to all tweets sharing the same event label by performing a majority vote:

$$\forall t \in T_{\text{train}} \colon \gamma_t = \arg\max_{\gamma} \left| \{ t' \colon \gamma_{t'} = \gamma \wedge E_{t'} \cap E_t \neq \emptyset \} \right|. \tag{2.6}$$

This way every tweet in the training set is represented by a semantic topic that is dominated on the level of the events instead of on tweet level, resulting in a much coarser attribution of semantic labels. We cannot do this for the test set, since we do not know the event labels for the test set while executing the algorithm. We can however try to emulate such a majority voting at runtime. For this purpose, every tweet t is associated with a set of semantic topics Γ_t. We initialize this set as follows:

$$\forall t \in T_{\text{test}} \colon \Gamma_t = \{\gamma_t\}. \tag{2.7}$$

Next to a set of topics for every tweet, we consider a dedicated hashtag pool H_h for every hashtag h, by analogy with [19]. With every pool H we associate a single semantic topic β_H. As the algorithm proceeds, more and more hashtag pools will be created and filled with tweets.

When a tweet t is processed in the clustering algorithm, it will first be added to some hashtag pools, depending on the number of hashtags in t. So for every hashtag h in t, t is added to H_h. When a tweet t is added to a hashtag pool H, a majority vote inside this pool is performed:

$$\beta_{\text{new},H} = \arg\max_{\gamma} \left| \{ t' \colon t' \in H \wedge \gamma_{t'} = \gamma \} \right|. \tag{2.8}$$

We then update Γ_t for every tweet t in H:

$$\forall t \in H \colon \Gamma_{\text{new},t} = (\Gamma_{\text{old},t} \setminus \{\beta_H\}) \cup \{\beta_{\text{new},H}\}. \tag{2.9}$$

Finally $\beta_{\text{new},H}$ becomes the new semantic topic of H. Note that every tweet t keeps its original semantic topic γ_t.

What still needs adjustment in order for the clustering algorithm to use this new information, is the definition of the semantic fraction from Equation (2.4). We altered the definition as follows:

$$\sigma'(t, S) = \max_{g \in \Gamma_t} \frac{|\{ t' \colon t' \in S \wedge g \in \Gamma_{t'} \}|}{|S|}. \tag{2.10}$$

Since Equation (2.10) implies Equation (2.4) if Γ_t contains only one element for every tweet t, this is a justifiable generalization.

2.4 Data collection and processing

In the past many datasets have been assembled to perform event clustering on social media. Unfortunately many of these datasets are not publicly available; this is especially true for Twitter datasets. We therefore choose to build our own dataset. To speed up this task we follow a semi-manual approach, in which we first collect candidate events based on a hashtag clustering procedure, after which we manually verify which of these correspond to real-world events.

2.4.1 Event definition

To identify events in a dataset consisting of thousands of tweets, we state the following event definition, which consists of three assumptions.

ASSUMPTION 1 – a real-world event is characterized by one or multiple hashtags. For example, tweets on the past FIFA world cup football matches were often accompanied by hashtags such as #USAvsBelgium and #WorldCup.

ASSUMPTION 2 – the timespan of an event cannot transgress the boundaries of a day. This means that if a certain real-world event takes place at several days—such as a music festival—this real-world event will be represented by multiple event labels. The assumption will allow us to discern events that share the same hashtag, but occur on a different day of the week, and will speed up the eventual event detection process. The hashtag #GoT for example will spike in volume whenever a new episode of Game of Thrones is aired, which are thus different events according to our definition.

ASSUMPTION 3 – there is only one event that corresponds to a certain hashtag on a given day.

Assumption 3 is not restrictive and can easily be relaxed. For example if we would relax this Assumption and allow multiple events with the same hashtags to happen on the same day, we would need a feature in the event detection process to incorporate time differences, which is easily done. Alternatively we could represent our tweets using df-idf$_t$ vectors, instead of tf-idf vectors, which also consider time aspects of the tweets [1].

2.4.2 Collecting data

We assembled a dataset by querying the Twitter Streaming API for two weeks, between September 29 and October 13 of the year 2014. We used a geolocation query and required that the tweets originated from within the Flanders region in Belgium, at least by approximation. Since only very few tweets are geotagged, our dataset was far from a representative sample of the tweets sent during this fortnight.

We therefore augment our dataset to make it more representative for an event detection task. If a real-world event is represented by one or more hashtags (Assumption 1), then we assume that at least one tweet with these hashtags is geotagged and that these hashtags are therefore already present in the original dataset. We thus consider every hashtag in the original dataset and use them one by one to query the Twitter REST API.

A query to the REST API returns an ordered batch of tweets $(t_i)_{i=1}^m$, where m is at most 100. By adjusting the query parameters—e.g. the maximum ID of the tweets—one can use multiple requests to gather tweets up to one week in the past. To make sure we only gather tweets from within Flanders, the tokens in the user location text field of every tweet in the current batch are compared to a list of regions, cities, towns and villages in Flanders, assembled using Wikipedia and manually adjusted for multilingual support. If the user location field is empty, the tweet is not considered further. We define a batch $(t_i)_{i=1}^m$ to be valid if and only if

$$\frac{|\{t_i \colon t_i \text{ in Flanders}\}|}{\text{timestamp}(t_m) - \text{timestamp}(t_1)} > \tau_1, \tag{2.11}$$

where τ_1 is a predefined threshold. If there are τ_2 subsequent invalid batches, all batches for the current considered hashtag are discarded. If there are τ_3 batches in total for which less than τ_4 tweets were sent in Flanders, all batches for the current considered hashtag are discarded as well. If none of these rules apply, all batches for the current hashtag are added to the dataset. When the timestamp$(\cdot)$ function is expressed in minutes, we set $\tau_1 = 1$, $\tau_2 = 12$, $\tau_3 = 25$ and $\tau_4 = 10$, as this yielded a good trade-off between execution time and quality of the data.

2.4.3 Collecting events

Using the assembled data and the event definition of Section 2.4.1 we can assemble a ground truth for event detection in three steps. Since events are represented by one or more hashtags according to Assumption 1, we first cluster the hashtags in the tweets using a co-occurrence measure. Next we

determine whether such a cluster represents an event, and finally we label the tweets corresponding with this cluster with an appropriate event label.

To assemble frequently co-occurring hashtags into clusters, a so-called co-occurrence matrix is constructed. It is a three-dimensional matrix Q that holds information on how many times two hashtags co-occur in a tweet. Since events can only take place on one day (Assumption 2), we calculate co-occurrence on a daily basis. If hashtag k and hashtag ℓ co-occur $a_{k,\ell,d}$ times on day d, then

$$\forall k, \ell, d \colon Q_{k,\ell,d} = \frac{a_{k,\ell,d}}{\sum_i a_{k,i,d}}. \tag{2.12}$$

To cluster co-occurring hashtags we adopt the standard DBSCAN clustering algorithm. This is an online clustering algorithm that requires two thresholds to be set: the minimum number of hashtags $\min_h$ per cluster and a minimum similarity measure ϵ between two hashtags above which the two hashtags reside in the same ϵ-neighbourhood. The similarity measure between hashtags k and ℓ on day d is defined as

$$\mathrm{sim}_{k,\ell,d} = \frac{Q_{k,\ell,d} + Q_{\ell,k,d}}{2}. \tag{2.13}$$

If we run DBSCAN for every day in the dataset, we obtain a collection of clusters of sufficiently co-occurring hashtags on the same day.

A lot of these clusters however do not represent a real-world event. Hashtags such as #love or #followme do not exhibit event-specific characteristics, such as an isolated, statistically significant peak in tweet volume per minute, but can rather be seen as near-constant noise in the Twitter feed. In order to identify the hashtags that do represent events and to filter out the noise, we follow a peak detection strategy. For this purpose we treat each cluster of hashtags separately, and we refer to the hashtags in these clusters as 'event hashtags'. With each cluster C we associate all the tweets that were sent on the same day and that contain one or more of the event hashtags in this cluster. We gather them in a set T_C. After sorting the tweets in T_C according to their timestamp, we calculate how many tweets are sent in every timeslot of five minutes, which makes up for a sequence $(v_{C,i})_{i=1}^{n}$ of tweet volumes, with n the number of time slots. We define that some v_{C,i^*} is an isolated peak in the sequence $(v_{C,i})$ if and only if

$$v_{C,i^*} \geq \theta_1 \wedge \forall i \neq i^* \colon v_{C,i^*} \geq v_{C,i} + \theta_2, \tag{2.14}$$

with θ_1 and θ_2 predefined thresholds. Only if one such isolated peak exists (Assumption 3), we label all tweets t in T_C with the same unique event label e_t and add them to the ground truth. Since we used the event hashtags

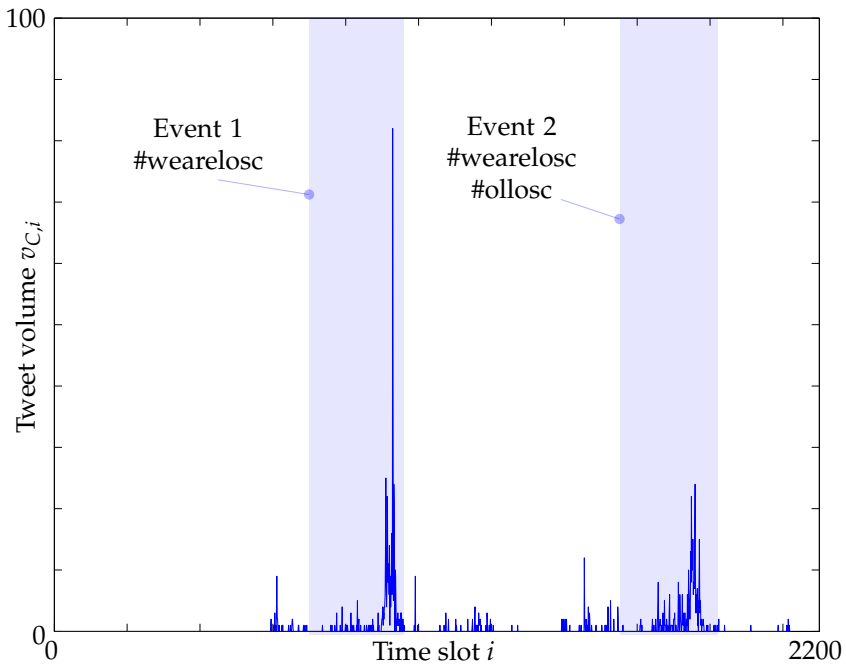


Figure 2.1: Plot of tweet volume as a function of time slot for two example events in the dataset, with their associated hashtags.

from C to construct this event, we have to remove all event hashtags in C from the tweets in T_C, otherwise the tweets themselves would already reflect the nature of the events in the ground truth.

With this procedure it is however likely that some tweets will belong to multiple events, but only get one event label. This is possible if a tweet contains multiple event hashtags that belong to different event hashtag clusters. We therefore alter the ground truth in which every tweet t corresponding to an event is associated with a set of event labels E_t instead of only one label. Of course, for the majority of these tweets, this set will only contain one event label.

In our final implementation we set $\min_{ll} = 1$, $\epsilon = 0.3$, $\theta_1 = 10$ and $\theta_2 = 5$. With these parameters clusters of co-occurring hashtags are rarely bigger than three elements. After manual inspection and filtering, the final dataset contains 322 different events adding up to a total of 63,067 tweets (ca. 12% of the original dataset). We assign $2/3$ of the events to a training set and $1/3$ to a test set, leading to 29,844 tweets in the training set and 33,223 in the test set.

Figure 2.1 shows a plot of the tweet volume in function of time slot for

two events in the dataset. The plot only covers the first week in the dataset. The events are two football games of the French team LOSC Lille—which is a city very near Flanders, and therefore shows up in our dataset. The first event is characterised by the single hashtag #wearelosc, and the second event by two hashtags: #wearelosc and #ollosc. Our algorithm detects the peaks in tweet volume during the games, and since only one significant peak exists per day, we assign the same event label to all tweets with the associated hashtags sent during that day.

2.5 Results

2.5.1 Performance measures

To assess the performance of the clustering algorithms, we report our results in terms of precision P, recall R and F_1 measure, as defined in [3, 9], and restated here:

$$P = \frac{1}{|T|} \sum_{t \in T} \frac{|S_t \cap \{t' : e_{t'} = e_t\}|}{|S_t|}, \tag{2.15}$$

$$R = \frac{1}{|T|} \sum_{t \in T} \frac{|S_t \cap \{t' : e_{t'} = e_t\}|}{|\{t' : e_{t'} = e_t\}|}, \tag{2.16}$$

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}, \tag{2.17}$$

in which T stands for the total dataset of tweets. When tweets can have multiple event labels, these definitions however do not apply any more. We therefore alter them as follows:

$$P = \frac{1}{|T|} \sum_{t \in T} \max_e \frac{|S_t \cap \{t' : e \in E_{t'} \wedge e \in E_t\}|}{|S_t|}, \tag{2.18}$$

$$R = \frac{1}{|T|} \sum_{t \in T} \max_e \frac{|S_t \cap \{t' : e \in E_{t'} \wedge e \in E_t\}|}{|\{t' : e \in E_{t'} \wedge e \in E_t\}|}. \tag{2.19}$$

Note that Equations (2.18) and (2.19) imply Equations (2.15) and (2.16) if there is only one event label per tweet.

We will also use purity as an indicator of the quality of the event clusters we obtain. We have chosen the definition of purity as in [22] and adapted it to our context as follows:

$$\text{purity} = \frac{1}{|T|} \sum_{t \in T} \max_e \frac{|S_t \cap \{t' : e = e_{t'}\}|}{|S_t|}. \tag{2.20}$$

It is a measure that is closely related to precision.

Table 2.1: Using hashtag-level semantics clearly outperforms baseline and plain semantics-driven clustering.

	Precision	Recall	F_1-measure
Baseline	47.12%	35.35%	40.40%
Semantics-driven	52.80%	30.60%	38.74%
Hashtag semantics	48.62%	36.97%	42.00%
Baseline (multi)	64.96%	36.36%	46.62%
Semantics-driven (multi)	69.27%	31.47%	43.28%
Hashtag semantics (multi)	64.06%	37.77%	47.52%

For multiple event labels, we alter this measure as follows:

$$\text{purity} = \frac{1}{|T|} \sum_{t \in T} \max_{e} \frac{|S_t \cap \{t' : e \in E_{t'}\}|}{|S_t|}. \tag{2.21}$$

2.5.2 Results

We now discuss the results of the algorithms explained in Section 2.3 with the use of the dataset constructed in Section 2.4. In the algorithms we make use of a set A of active event clusters, which become inactive after some time period. We could for example use an exponential decay function to model the time after which a cluster becomes inactive since the last tweet was added. Using Assumption 2 however we can use a much simpler method: when a new day begins, all event clusters are removed from A and thus become inactive. This way we start with an empty set A of active clusters every midnight.

For the semantics-driven clustering algorithm we assign the tweets to 10 TwitterLDA topics using the standard parameters proposed in [21] and 500 iterations of Gibbs sampling. Table 2.1 shows the results of the baseline algorithm, the semantics-driven algorithm and the hashtag-level semantics approach, both for one event label and multiple event labels per tweet. Note that, since we have removed the event hashtags from the tweets in the ground truth, the hashtag-level semantics approach does not use any implicit or explicit information about the nature of the events.

We note that the hashtag-level semantics approach outperforms the baseline clustering algorithm, with an increase of 1.6 percentage points in F_1-measure for single event labels. In terms of precision and recall, hashtag-level semantics performs better in both metrics than baseline (significant improvement, $p < 0.001$ in t-test). When using multiple event labels per tweet, precision is decreased by 0.9 percentage points, but raises recall with 1.4 percentage points, leading to an increase of F_1-measure by 0.9 percentage points.

Compared to the standard semantics-driven algorithm we do 6 percentage points better in recall, but 4 percentage point worse in precision for single event labels. Hashtag-level semantic clustering seems to manage to account for the substantial loss in recall that occurs when using the basic semantics-driven method, but lacks in precision; the precision is however still 1.5 percentage points better than the baseline algorithm. The plain semantics-driven approach is 1.7 percentage points worse than baseline in terms of F_1-measure, but provides much more precision by sacrificing in recall. For multiple event labels the differences are even more pronounced between the standard semantics approach and the other algorithms. The former performs 3.3 percentage points worse in F_1-measure compared to baseline, and 4.2 percentage points worse compared to hashtag semantics. Using multiple event labels, the plain semantics-driven algorithm however has a much higher precision than baseline and hashtag semantics.

To assess the significance of the differences in F_1 measure between our three systems, we used a Bayesian technique suggested by Goutte et al. [23]. First we estimated the true positive, false positive and false negative numbers for the three systems. Next we sampled 10,000 gamma variates from the proposed distribution for F_1 for these systems and calculated the probability of one system being better than another system. We repeated this process 10,000 times. Hashtag semantics resulted in a higher F_1 measure in 99.99% of the cases; our results are thus a significant improvement over baseline. By contrast, the plain semantics-driven approach is significantly worse than baseline, also in 99.99% of the cases. Concerning multiple event labels, the hashtag semantics approach is better in 98.5% of the cases than baseline, which is also a significant improvement—although less than in the single event label case.

We also compare our three approaches in terms of cluster purity and the number of detected event clusters. These numbers are shown in Table 2.2. We see that the purity of the clusters in the plain semantics-driven approach is higher than baseline and hashtag semantics, but the number of detected event clusters is even substantially larger. This explains the high precision and low recall of the semantics-driven algorithm. The purity of baseline and hashtag semantics is almost equal, but the latter approach discerns more events than baseline, thereby explaining the slight increase in precision and recall for the hashtag semantics approach compared to baseline. Concerning multiple event labels, the purity increases significantly compared to single event labels. Since the number of detected events remains the same, this explains the substantial increase in precision for the multi-label procedure.

Table 2.2: A comparison of baseline, plain semantics-driven clustering and hashtag semantics in terms of purity and number of event clusters.

	Purity	Number of events
Baseline	61.29%	409
Semantics-driven	64.76%	662
Hashtag semantics	61.15%	441
Baseline (multi)	75.51%	409
Semantics-driven (multi)	77.74%	662
Hashtag semantics (multi)	73.72%	441

2.6 Conclusion

We developed two semantics-based extensions to the single-pass baseline clustering algorithm as used by Becker et al. to detect events in Twitter streams. In this we used semantic information about the tweets to drive the event detection. For this purpose we assigned a topic label to every tweet using the TwitterLDA algorithm. To evaluate the performance of the algorithms we semi-automatically developed a ground truth using a hashtag clustering and peak detection strategy, to aid the manual labelling of tweets with events. When using the topic labels at the level of individual tweets, the algorithm performs significantly worse than baseline. When however gathering the semantic labels of the tweets on a coarser, hashtag level we get a significant gain over baseline. We can conclude that high-level semantic information can indeed improve new and existing event detection and clustering algorithms.

References

[1] L. M. Aiello, G. Petkos, C. Martin, D. Corney, S. Papadopoulos, R. Skraba, A. Goker, I. Kompatsiaris, and A. Jaimes. *Sensing Trending Topics in Twitter*. Multimedia, IEEE Transactions on, 2013.

[2] H. Becker, M. Naaman, and L. Gravano. *Beyond Trending Topics: Real-World Event Identification on Twitter.* In ICWSM 2011: International AAAI Conference on Weblogs and Social Media, 2011.

[3] T. Reuter and P. Cimiano. *Event-based classification of social media streams*. In ICMR '12: Proceedings of the 2nd ACM International Conference on Multimedia Retrieval, 2012.

[4] T. Sakaki, M. Okazaki, and Y. Matsuo. *Earthquake shakes Twitter users: real-time event detection by social sensors*. In WWW '10: Proceedings of the 19th international conference on World wide web, 2010.

[5] G. Stilo and P. Velardi. *Time Makes Sense: Event Discovery in Twitter Using Temporal Similarity*. In Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on, 2014.

[6] J. Yin, A. Lampert, M. Cameron, B. Robinson, and R. Power. *Using social media to enhance emergency situation awareness*. IEEE Intelligent Systems, 2012.

[7] S. Van Canneyt, S. Schockaert, and B. Dhoedt. *Estimating the Semantic Type of Events Using Location Features from Flickr*. In SIGSPATIAL '14, 2014.

[8] H. Becker, M. Naaman, and L. Gravano. *Event Identification in Social Media.* In WebDB 2009: Twelfth International Workshop on the Web and Databases, 2009.

[9] H. Becker, M. Naaman, and L. Gravano. *Learning similarity metrics for event identification in social media*. In WSDM '10: Third ACM international conference on Web search and data mining, 2010.

[10] S. Petrović, M. Osborne, and V. Lavrenko. *Streaming first story detection with application to Twitter*. In HLT '10: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, 2010.

[11] J. Nichols, J. Mahmud, and C. Drews. *Summarizing sporting events using twitter*. In IUI '12: Proceedings of the 2012 ACM international conference on Intelligent User Interfaces, 2012.

[12] F. Chierichetti, J. Kleinberg, R. Kumar, M. Mahdian, and S. Pandey. *Event Detection via Communication Pattern Analysis*. In ICWSM '14: International Conference on Weblogs and Social Media, 2014.

[13] J. Weng, Y. Yao, E. Leonardi, and B.-S. Lee. *Event Detection in Twitter.* In ICWSM '11: International Conference on Weblogs and Social Media, 2011.

[14] L. Chen and A. Roy. *Event detection from flickr data through wavelet-based spatial analysis*. In CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management, 2009.

[15] A. Ritter, Mausam, O. Etzioni, and S. Clark. *Open domain event extraction from twitter*. In KDD '12: Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, 2012.

[16] D. M. Blei, A. Y. Ng, and M. I. Jordan. *Latent dirichlet allocation*. Machine Learning, 2003.

[17] Y. Liu, A. Niculescu-Mizil, and W. Gryc. *Topic-link LDA: joint models of topic and author community*. In ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning, 2009.

[18] Y. Wang, E. Agichtein, and M. Benzi. *TM-LDA: efficient online modeling of latent topic transitions in social media*. In KDD '12: Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, 2012.

[19] R. Mehrotra, S. Sanner, W. Buntine, and L. Xie. *Improving lda topic models for microblogs via tweet pooling and automatic labeling*. 2013.

[20] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. *LIBLINEAR: A Library for Large Linear Classification*. The Journal of Machine Learning Research, 2008.

[21] W. X. Zhao, J. Jiang, J. He, Y. Song, P. Achananuparp, E.-P. Lim, and X. Li. *Topical keyphrase extraction from Twitter*. In HLT '11: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, 2011.

[22] C. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[23] C. Goutte and E. Gaussier. *A probabilistic interpretation of precision, recall and F-score, with implication for evaluation*. In ECIR'05: Proceedings of the 27th European conference on Advances in Information Retrieval Research, 2005.

3

Representation Learning for Very Short Texts using Weighted Word Embedding Aggregation

"The only formula I know will work for us is that when we're together in the sum of our parts, it's far greater than what we added up to at the start."

—Little Boots, 2009

In Chapter 2 we have represented tweets by a tf-idf vector. Using such a representation, we count the number of overlapping words between two tweets to determine the similarity between them. If more words overlap, the tweets are more similar to each other. However, since the number of words in a tweet is very limited, the probability of one or more words overlapping quickly drops. Spelling mistakes and common word corruptions, as we have discussed in Chapter 1, make things even worse. We have also seen that word embeddings are able to model the semantic similarity between words, even though they are different. Therefore, we will investigate a method of representing tweets and other short texts based on word embeddings, thereby alleviating the problem of word overlap. The challenge lies in effectively combining the embeddings of all words in a tweet to calculate a semantically-rich and high-quality tweet embedding.

⋆ ⋆ ⋆

C. De Boom, S. Van Canneyt, T. Demeester, and B. Dhoedt.

Appeared in Pattern Recognition Letters, 80:150–156, 2016.

Abstract Short text messages such as tweets are very noisy and sparse in their use of vocabulary. Traditional textual representations, such as tf-idf, have difficulty grasping the semantic meaning of such texts, which is important in applications such as event detection, opinion mining, news recommendation, etc. We constructed a method based on semantic word embeddings and frequency information to arrive at low-dimensional representations for short texts designed to capture semantic similarity. For this purpose we designed a weight-based model and a learning procedure based on a novel median-based loss function. This paper discusses the details of our model and the optimization methods, together with the experimental results on both Wikipedia and Twitter data. We find that our method outperforms the baseline approaches in the experiments, and that it generalizes well on different word embeddings without retraining. Our method is therefore capable of retaining most of the semantic information in the text, and is applicable out-of-the-box.

3.1 Introduction

Short pieces of text reach us every day through the use of social media such as Twitter, newspaper headlines, and texting. Especially on social media, millions of such short texts are sent every day, and it quickly becomes a daunting task to find similar messages among them, which is at the core of applications such as event detection [1], news recommendation [2], etc.

In this paper we address the issue of finding an effective vector representation for a very short text fragment. By effective we mean that the representation should grasp most of the semantic information in that fragment. For this we use semantic word embeddings to represent individual words, and we learn how to weigh every word in the text through the use of tf-idf (term frequency - inverse document frequency) information to arrive at an overall representation of the fragment.

These representations will be evaluated through a semantic similarity task. It is therefore important to point out that textual similarity can be achieved on different levels. At the most strict level, the similarity measure between two texts is often defined as being (near) paraphrases. In a more relaxed setting one is interested in topic- and subject-related texts. For example, if a sentence is about the release of a new Star Wars episode and another about Darth Vader, they will be dissimilar in the most strict

sense, although they share the same underlying subject. In this paper we focus on the broader concept of topic-based semantic similarity, as this is often applicable in the already mentioned use cases of event detection and recommendation.

Our main contributions are threefold. First, we construct a technique to calculate effective text representations by weighing word embeddings, for both fixed- and variable-length texts. Second, we devise a novel median-based loss function to be used in the context of minibatch learning to mitigate the negative effect of outliers. Finally we create a dataset of semantically related and non-related pairs of text from both Wikipedia and Twitter, on which the proposed techniques are evaluated.

We will show that our technique outperforms most of the baselines in a semantic similarity task. We will also demonstrate that our technique is independent of the word embeddings being used, so that the technique is directly applicable and thus does not require additional model training when used in different contexts, in contrast to most state-of-the art techniques.

In the next section, we start with a summary of the related work, and our own methodology will be devised in Section 3.3. Next we explain how data is collected in Section 3.4, after which we discuss our experimental results in Section 3.5.

3.2 Related work

In this work we use so-called word embeddings as a basic building block to construct text representations. Such an embedding is a distributed vector representation of a single word in a fixed-dimensional semantic space, as opposed to term tf-idf vectors, in which a word is represented by a one-hot vector [3, 4]. A word's term frequency (tf) is the number of times the word occurs in the considered document, and a word's document frequency (df) is the number of documents in the considered corpus that contain that word. Its (smoothed) inverse document frequency (idf) is defined as:

$$\text{idf} \triangleq \log \frac{N}{1 + \text{df}}, \tag{3.1}$$

in which N is the number of documents in the corpus [4]. A tf-idf-based similarity measure is based on exact word overlap. As texts become smaller in length, however, the probability of having words in common decreases. Furthermore, these measures ignore synonyms and any semantic relatedness between different words, and are prone to negative effects of homonyms.

Instead of relying on exact word overlap, one can incorporate semantic information into the similarity process. Latent Semantic Indexing (LSI) and Latent Dirichlet Allocation (LDA) are two examples, in which every word is projected into a semantic (topic) space [5, 6]. At test time, inference is performed to obtain a semantic vector for a particular sentence. Both training and inference of standard LSI and LDA, however, are computationally expensive on large vocabularies.

Although LSI and LDA have been used with success in the past, Skip-gram models have been shown to outperform them in various tasks [7, 8]. In Skip-gram, part of Google's word2vec toolkit[1], distributed word embeddings are learned through a neural network architecture to predict its surrounding words in a fixed window.

Once the word embeddings are obtained, we have to combine them into a useful sentence representation. One possibility is to use an multilayer perceptron (MLP) with the whole sentence as an input, or a 1D convolutional neural network [9–12]. Such an approach, however, requires either an input of fixed length or aggregation operations—such as dynamic k-max pooling [13]—to arrive at a sentence representation that has the same dimensionality for every input. Recurrent neural networks (RNNs) and variants can overcome the problem of fixed dimensionality or aggregation, since one can feed word after word in the system and in the end arrive at a text representation [14–16]. The recently introduced Skip-thought vectors, heavily inspired on Skip-gram, combine the learning of word embeddings with the learning of a useful sentence representation using an RNN encoder and decoder [17]. RNN-based methods present a lot of advantages over MLPs and convolutional networks, but still retraining is required when using different types of embeddings.

Paragraph2vec is another method, inspired by the Skip-gram algorithm, to derive sentence vectors [18]. The technique requires the user to train vectors for frequently occurring word groups. The method, however, is not usable in a streaming or on-the-fly fashion, since it requires retraining for unseen word groups at test time.

Aggregating word embeddings through a mean, max, min... function is still one of the most easy and widely used techniques to derive sentence embeddings, often in combination with an MLP or convolutional network [9, 19–21]. On the one hand, the word order is lost, which can be important in e.g. paraphrase identification. On the other hand, the methods are simple, out-of-the-box and do not require a fixed length input.

Related to the concepts of semantic similarity and weighted embedding aggregation, there is extensive literature. In [22], Kusner et al. calculate a

[1] Available at code.google.com/archive/p/word2vec

similarity metric between documents based on the travel distance of word embeddings from one document to another one. We on the other hand will derive vectors for the documents themselves. Kenter et al. learn semantic features for every sentence in the dataset based on a saliency weighted network for which the Okapi BM25[2] algorithm is used [23]. However, the features are being learned for every sentence prior to test time, and therefore not applicable in a real-time streaming context. Finally, in [24] Kang et al. calculate a cosine similarity matrix between the words of two sentences that are sorted based on their idf value, which they use as a feature vector for an MLP. Their approach is similar to our work in the sense that the authors use idf information to rescale term contribution. Their primary goal, however, is calculating semantic similarity instead of learning a sentence representation. In fact, the authors totally discard the original word embeddings and only use the calculated cosine similarity features.

3.3 Methodology

The core principle of our methodology is to assign a weight to each word in a short text. These weights are determined based on the idf value of the individual words in that text. The idea is that important words—i.e. words that are needed to determine most of the text's semantics—usually have higher idf values than less important words, such as articles and auxiliaries... Indeed, the latter appear more frequently in various different texts, while words with a high idf value mostly occur in similar contexts. The final goal is to combine the weighted words into a semantically effective, single text representation.

To achieve this goal, we will model the problem of finding a suitable text representation as a semantic similarity task between couples of short texts. In order to classify such couples of text fragments into either semantically related pairs or non-related pairs, the vector representations of these two texts are directly compared. In this paper we use a simple threshold function on the distance between the two text representations, as we want related pairs to lie close to each other in their representation space, and non-related pairs to lie far apart:

$$g(t_1, t_2) = \begin{cases} \text{pair} & \text{if } d(t_1, t_2) \leq \theta \\ \text{non-pair} & \text{if } d(t_1, t_2) > \theta \end{cases}. \tag{3.2}$$

In this expression t_1 and t_2 are two short text vector representations of dimensionality ν, $d\colon (x, y) \in \mathbb{R}^{2\nu} \rightarrow \mathbb{R}^+$ is a vector distance function of

[2]BM25 is a scoring function—as an alternative to tf-idf and cosine similarity—that uses term frequencies as well as the inverse document frequency to rank documents given a query.

choice (e.g. cosine distance, euclidean distance...), θ is a threshold, and $g(\cdot)$ is the binary prediction of semantic relatedness.

3.3.1 Basic architecture

As mentioned before, we will assign a weight to each word in a text according to that word's idf value. To learn these weights, we devise a model that is visualised in Figure 3.1. In the learning scheme, we use related and non-related couples of text as input data. First, the words in every text are sorted from high to low idf values. Original word order is therefore discarded, as is the case in usual standard aggregation operations. After that, every embedding vector for each of the sorted words is multiplied with a weight that can be learned. Finally, the weighted vectors are averaged to arrive at a single text representation.

In more detail, consider a dataset $\mathcal{D}$ consisting of couples of short texts. An arbitrary couple is denoted by C, and the two texts of C by C^α and C^β. We indicate the vector representation of word j in text C^α by C^α_j. All word vectors have the same dimensionality ν. Each text C^α also has an associated length $n(C^\alpha)$, i.e. the number of words in C^α. For now, in this section, we assume that $n \triangleq n(C^\alpha) = n(C^\beta), \forall C \in \mathcal{D}$, a notion we will relax in Section 3.3.3. The final goal of our methodology is to arrive at a vector representation for both texts in C, denoted by $t(C^\alpha)$ and $t(C^\beta)$. Denoting the sorted texts as $C^{\alpha'}$ and $C^{\beta'}$, we arrive at a vector representation $t(C^\alpha)$ and $t(C^\beta)$ through the following equation:

$$\forall \ell \in \{\alpha, \beta\} : t(C^\ell) = \frac{1}{n} \sum_{j=1}^{n} w_j \cdot C_j^{\ell'}, \tag{3.3}$$

in which $w_j, j \in \{1, \ldots, n\}$ are the weights to be learned. As such, we create a weighted sum of the individual embeddings, the weights of which are only related to the rank order according to the idf values of all words in the fragment.

The model we construct through this procedure is related to the siamese neural network with parameter sharing from the early nineties [25]. The learning procedure of such models is as follows. We first calculate the vector representations for both texts in a particular couple through Equation (3.3), both using the same weights, after which we compare the two vector representations through a loss function $\mathcal{L}\left(t(C^\alpha), t(C^\beta)\right)$ that we wish to minimize. After that, the weights are updated through a gradient descent procedure using a learning rate η:

$$\forall j \in \{1, \ldots, n\} : w_j \leftarrow w_j - \eta \cdot \frac{\partial}{\partial w_j} \mathcal{L}\left(t(C^\alpha), t(C^\beta)\right). \tag{3.4}$$

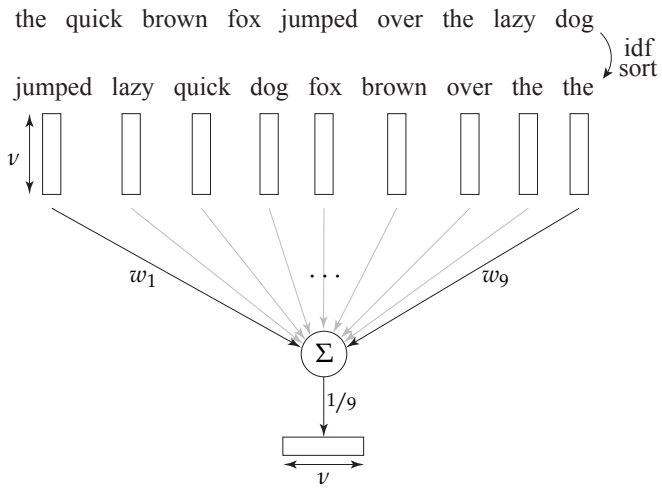


Figure 3.1: Illustration of the weighted average approach for a toy sentence of nine words long and word vectors of dimension ν.

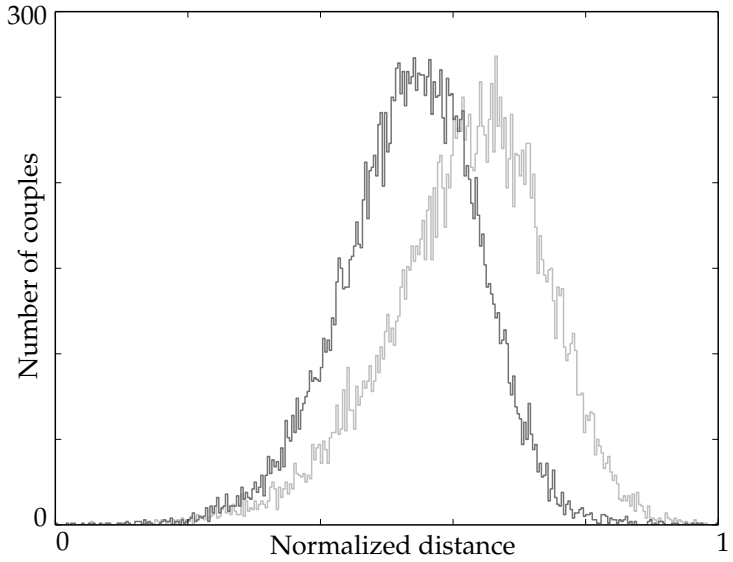


Figure 3.2: Example distributions of distances between non-related pairs (light grey) and related pairs (dark grey) on Twitter data, using a simple mean of word embeddings.

3.3.2 Loss functions

As pointed out in the beginning of this section, we want to have semantically related texts to lie close to each other in the representation space, and non-related texts to lie far apart from each other. We can visually inspect the distribution of the distances between every couple in the dataset. In fact, we calculate two distributions, one for the related pairs and one for the non-related pairs. Two examples of such distributions, created using Twitter data and an average word embedding text representation, are shown in Figure 3.2. Related pairs tend to lie closer to each other than non-related pairs. There is however a considerable overlap between the two distributions. This means that making binary decisions on similarity based on a well-chosen distance threshold will lead to a large error. The goal is to reduce this error, and thus to minimize the overlap between the two distributions. Directly minimizing the overlap is difficult, since it requires the overlap as a function of the model weights. We will instead describe two different loss functions as an approximation to this problem.

The first loss function is related to the contrastive loss function regularly used in siamese neural architectures [26]. We define the quantity p_C as follows:

$$p_C \triangleq \begin{cases} 1 & \text{if } C \text{ is a related pair,} \\ -1 & \text{if } C \text{ is a non-related pair.} \end{cases} \tag{3.5}$$

The loss function, which we will conveniently call the contrastive-based loss, is then given by:

$$\mathcal{L}_c \left(t(C^\alpha), t(C^\beta) \right) = p_C \cdot d \left(t(C^\alpha), t(C^\beta) \right), \tag{3.6}$$

in which $d \colon (x, y) \in \mathbb{R}^{2\nu} \to \mathbb{R}^+$ is a vector distance function of choice, as before. Note that, when trying to minimize this loss, related pairs will get pushed to each other in the representation space, while non-related pairs will get dragged apart.

This loss function, however, has two main problems. First, there is an imbalance between the loss for related pairs and non-related pairs, in which the latter can get an arbitrarily negative loss, while the related pairs' loss cannot be pushed below zero. To solve this, we could add a maximum possible distance—which is e.g. 1 in the case of cosine distance—but this cannot be generalized to arbitrary distance functions. Second, this loss function can skew distance distributions, so that minimizing overlap between distributions is not guaranteed. In fact, the overlap can even increase while minimizing this loss. This happens, for example, when the distance between some of the related pairs can be drastically reduced while

other related pairs get dragged farther apart, and vice versa for the non-related pairs. The loss function as such allows this to happen, since it can focus on data points that are easier to shift towards or away from each other and it can ignore what happens to the other data points. As long as the contribution of these shifts to the overall loss remains dominant, the loss will diminish, although the predictions according to Equation (3.2) will be worse. Despite these problems we still consider this loss function due to its simplicity. The derivative with respect to weight w_j is given by:

$$\frac{\partial}{\partial w_j}\mathcal{L}_{\mathrm{c}}\left(t(C^{\alpha}),t(C^{\beta})\right) = \frac{p_C}{n}\left(\nabla d(x,y)\right)\Big|_{x=\frac{1}{n}\sum_{j=1}^{n} w_j C_j^{\alpha'},\, y=\frac{1}{n}\sum_{j=1}^{n} w_j C_j^{\beta'}} \begin{bmatrix} C_j^{\alpha'} \\ C_j^{\beta'} \end{bmatrix}. \tag{3.7}$$

In a second loss function we try to mitigate the loss imbalance and potential skewing of the distributions caused by the contrastive-based loss function. For this purpose we will use the median, as it is a very robust statistic insensitive to outliers. As we need multiple data points to calculate the median, this loss function can only be used in the context of minibatch gradient descent, in which the number of positive and negative examples in each minibatch is balanced. In practice we consider a minibatch $B \subset \mathcal{D}$ of $n(B)$ randomly sampled data points, in which there are exactly $n(B)/2$ related pairs and $n(B)/2$ non-related pairs. We consider the couple of texts $M(B) \in B$ as the median couple if $\mu(B) = d(M(B)^{\alpha}, M(B)^{\beta})$ is the median of all distances between the couples in B:

$$M(B) \triangleq \arg\operatorname*{median}_{C\in B} d(C^{\alpha}, C^{\beta}); \tag{3.8}$$

$$\mu(B) \triangleq \operatorname*{median}_{C\in B} d(C^{\alpha}, C^{\beta}). \tag{3.9}$$

Since minibatch B is randomly sampled, we can consider $\mu(B)$ as an approximation to the optimal split point between related and non-related pairs, in the sense of threshold θ in Equation (3.2). We thus consider all related pairs with a distance larger than $\mu(B)$, and all non-related pairs with a distance smaller than $\mu(B)$ to be classified incorrectly. Since minimizing a 0-1 loss is NP-hard, we use a scaled cross-entropy function in our loss, which we will call the median-based loss:

$$\mathcal{L}_{\mathrm{m}}\left(t(C^{\alpha}),t(C^{\beta}),B\right) = \ln\left[1+\exp\left(-\kappa p_C\Big(\mu(B)-d\big(t(C^{\alpha}),t(C^{\beta})\big)\Big)\right)\right], \tag{3.10}$$

in which κ is a hyperparameter. The derivative with respect to weight w_j is given by the following expression (in which $\sigma(\cdot)$ is the sigmoid function):

$$\begin{aligned}
&\frac{\partial}{\partial w_j}\mathcal{L}_{\mathrm{m}}\left(t(C^{\alpha}), t(C^{\beta}), B\right) \\
&= \kappa\sigma\left(-\kappa p_C\left(\mu(B) - d\left(t(C^{\alpha}), t(C^{\beta})\right)\right)\right) \\
&\quad \cdot \frac{\partial}{\partial w_j}\left(\mathcal{L}_{\mathrm{c}}\left(t(C^{\alpha}), t(C^{\beta})\right) - \mathcal{L}_{\mathrm{c}}\left(t(M(B)^{\alpha}), t(M(B)^{\beta})\right)\right).
\end{aligned} \tag{3.11}$$

3.3.3 Texts with variable length

The method described thus far is only applicable to short texts of a fixed length, which is limiting. In this section we will extend our technique to texts of a variable, but given maximum length. For this purpose we have devised multiple approaches, of which we will elaborate the one that performed best in the experiments.

Suppose that all texts have a fixed maximum length of $n_{\max}$. In the learning procedure we will learn a total of $n_{\max}$ weights with the techniques described earlier. To find the weights for a text with length $m \leq n_{\max}$ we will use subsampling and linear interpolation. That is, for an arbitrary text C^{ℓ} we first find the sequence of real-valued indices $I(C^{\ell}, j), j \in \left\{1, \ldots, n(C^{\ell})\right\}$ through subsampling:

$$\forall j \in \left\{1, \ldots, n(C^{\ell})\right\} : I(C^{\ell}, j) \triangleq 1 + \frac{(j-1)(n_{\max}-1)}{n(C^{\ell})-1}. \tag{3.12}$$

Then, in the second step, we calculate the new weights $z_j(C^{\ell}), j \in \left\{1, \ldots, n(C^{\ell})\right\}$ for the words in C^{ℓ} through linear interpolation, in which ϵ is arbitrarily small:

$$\begin{aligned}
&\forall j \in \left\{1, \ldots, n(C^{\ell})\right\} : z_j(C^{\ell}) \triangleq \\
&\quad \frac{(w_{\lceil I_j(C^{\ell})\rceil} - w_{\lfloor I_j(C^{\ell})\rfloor})(I(C^{\ell}, j) - \lfloor I_j(C^{\ell})\rfloor)}{\lceil I_j(C^{\ell})\rceil - \lfloor I_j(C^{\ell})\rfloor + \epsilon} + w_{\lceil I_j(C^{\ell})\rceil}.
\end{aligned} \tag{3.13}$$

In this, $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ are resp. the ceil and floor functions. Finally, Equation (3.3) needs to be updated with these new weights instead of w_j:

$$\forall \ell \in \{\alpha, \beta\} : t(C^{\ell}) = \frac{1}{n(C^{\ell'})} \sum_{j=1}^{n(C^{\ell'})} z_j(C^{\ell'}) \cdot C_j^{\ell'}. \tag{3.14}$$

Calculating the derivative of the new weights with respect to the original weights is straightforward.

3.4 Data collection

To train the weights for the individual embeddings and to conduct experiments, we collect data from different sources. First we gather textual pairs from Wikipedia which we will use as a base dataset to finetune our methodology. We will also use this dataset to test our hypotheses and perform initial experiments. The second dataset will consist of Twitter message pairs, which we will use to show that our method can be used in a practical streaming context.

3.4.1 Wikipedia

We will perform our initial experiments in a base setting using English Wikipedia articles. The most important benefit of using Wikipedia is that there is a lot of well structured data. It is therefore a good starting point to collect a ground truth to finetune our methodology.

We use the English Wikipedia dump of March 4th 2015, and we remove its markup and punctuation. We convert all letters to lower case and every number is replaced by a single character '0' (zero). Next we construct related pairs of texts which both have the same, fixed length n. To do this, we take a Wikipedia article and we extract n consecutive words out of a paragraph. Then we skip two words, after which we extract the next n consecutive words, as long as they remain in the same paragraph. To extract non-related text pairs we follow the same procedure, but we make sure that the two texts are from different articles, which we choose at random. This approach is closely related to the data collection practice used in [10]. We want to emphasize again that our vision of semantic similarity is one of topic-based similarity instead of paraphrase-similarity, as discussed in the introduction. This notion is reflected in our data collection. We extract a total of 4.9 million related pairs and 4.9 million non-related pairs, each for fixed-length texts of 20 words long. We also extract 4.9 million related and non-related pairs of which the texts varies in length between 10 and 30 words. All datasets are divided into a train set of 1.5 million pairs, a test set of 1.5 million pairs and a validation set of 1.9 million pairs.

3.4.2 Twitter

Twitter is a very different kind of medium than Wikipedia. Wikipedia articles are written in a formal register and mostly consist of linguistically correct sentences. Twitter messages, on the other hand, count no more than 140 characters and are full of spelling errors, abbreviations and slang.

We propose that two tweets are semantically related if they are generated by the same event. As in [1], we require that such an event is represented by one or more hashtags. Since Twitter posts and associated events are very noisy by nature, we restrict ourselves to tweets by 100 English news agencies. We manually created this list through inspection of their Twitter accounts; the list is available through our GitHub page, see Section 3.6.

We gathered 341 949 tweets from all news agencies through the Twitter REST API at the end of August 2015. We convert all words to lowercase, replace all numbers by the single character '0' and remove non-informative hashtags such as *#breaking*, *#update* and *#news*. To generate related pairs out of these tweets, we consider four simple heuristic rules:

1. The number of words in each tweet, different from hashtags, mentions or URLs, should be at least 5.
2. The Jaccard similarity between the set of hashtags in both tweets should be at least 0.5.
3. The tweets should be sent no more than 15 minutes from each other.
4. The Jaccard similarity between the set of words in both tweets should be less than 0.5.

We add the last rule in order to have sufficient word dissimilarity between the pairs, as tweets that mostly contain the same words are too easy to relate. To generate non-related pairs, we remove rule 3 and rule 2 is changed: the Jaccard similarity between sets of hashtags should now be zero. Using these heuristics, we generate a train set of 15 000 pairs, a validation set of 20 000 pairs and a test set of 13 645 pairs, of which we remove all overlapping hashtags. We manually label 200 generated pairs and non-pairs, and we achieve an error rate of 28%. Due to the used heuristics and the linguistic nature of tweets in general, the ground truth can be considered very noisy; achieving an error rate lower than around 28% on this dataset will therefore be difficult, and the gain would not lead to a better model of the human notion of similarity anyway.

3.5 Experiments

In this section we discuss the results of several experiments on all aspects of our methodology given the data we collected. First we will discuss some results using the Wikipedia dataset, after which we also take a look at the Twitter dataset. We will use two performance metrics in our evaluation.

The first is the optimal split error, i.e. we classify all pairs according to Equation (3.2)—after determining the optimal split point θ—and we determine the classification error rate. A second performance metric is the Jensen-Shannon (JS) divergence. This is a symmetric distance measure between two probability distributions, related to the—well-known, but asymmetric—KL divergence. We will use it to capture the difference between the related and non-related pairs' distributions, as shown in Figure 3.2. The bigger the JS divergence, the greater the difference between the two distributions.

In our experiments we will use Google's *word2vec* software to calculate word embeddings. We choose Skip-gram with negative sampling as the learning algorithm, using a context window of five words and 400 dimensions. We feed an entire cleaned English Wikipedia dump of March 4th 2015 to the algorithm, after which we arrive at a total vocabulary size of 2.2 million words. Since we also need document frequencies, we calculate these for each of the vocabulary words using the same Wikipedia dump.

In previous work we showed that using an Euclidean distance function leads to a much better separation between related and non-related pairs than the more often used cosine distance, so we will also use the Euclidean distance throughout our experiments here [27]. Calculating the gradient of this distance function—which is used in Equation (3.7)—is straightforward.

To obtain the results hereafter, we use the following procedure. We use the train set to train the weights w_j in Equation (3.3). The validation set is used to determine the optimal split point θ in Equation (3.2). Finally predictions and evaluations are made on the test set. In the next two subsections we discuss the results on the Wikipedia and Twitter datasets.

3.5.1 Baselines

We will compare the performance of our methods to several baseline mechanisms that construct sentence vector representations. The simplest and most widely used baseline is a tf-idf vector. Comparing two tf-idf vectors is done through a standard cosine similarity. In a second baseline we simply take, for every dimension, the mean across all embeddings:

$$\forall \ell \in \{\alpha, \beta\} : \forall k \in \{1, \ldots, \nu\} : t(C^\ell)_k = \operatorname*{mean}_j C^\ell_{j,k}. \tag{3.15}$$

In the third baseline we replace the mean by a maximum operation. Taking a mean or a maximum is a very common approach to combine word embeddings—or other intermediary representations in an NLP

Table 3.1: Results for the Wikipedia data, for texts of length 20 and variable lengths, using Wikipedia and Google embeddings.

	Wikipedia, 20 words		Wikipedia, variable length		Google, variable length	
	Split error	JS divergence	Split error	JS divergence	Split error	JS divergence
Tf-idf	19.60%	0.3698	22.23%	0.3190	**22.50%**	**0.3130**
Mean	19.43%	0.3781	25.61%	0.2543	34.41%	0.1130
Max	19.05%	0.3981	27.09%	0.2308	37.05%	0.0842
Min/max	16.78%	0.4520	26.19%	0.2487	35.82%	0.1032
Mean, top 30% idf	17.02%	0.4435	22.67%	0.3182	32.07%	0.1512
Max, top 30% idf	18.05%	0.4193	28.32%	0.2150	36.38%	0.0999
Min/max, top 30% idf	16.40%	0.4581	27.86%	0.2260	35.61%	0.1140
Mean, idf-weighted	24.00%	0.2767	27.51%	0.2139	33.88%	0.1185
Learned weights, contrastive	14.44%	0.5080	21.20%	0.3503	30.50%	0.1776
Learned weights, median	**14.06%**	**0.5184**	**16.41%**	**0.4602**	25.10%	0.2641

architecture—into a sentence representation. We can also replace the maximum in Equation (3.15) by a minimum. The fourth baseline is a concatenation of the maximum and minimum vectors, resulting in a vector having two times the original dimensionality ('min/max'). We can also apply the three previous baselines—i.e. 'mean', 'max' and 'min/max'—only considering words with a high idf value ('top 30% idf'). That is, we sort the words in a text based on their idf values, and we take the mean or maximum of the top 30%. In a final baseline we weigh each word in the sentence with its corresponding idf value and then take the mean ('mean, idf-weighted').

3.5.2 Details on the learning procedure

In the results below we use the methodology from Section 3.3 to learn weights for individual words in a short text. All procedures are implemented with the Theano[3] framework and executed using an Nvidia Tesla K40 GPU. We use mini-batch stochastic gradient descent as learning procedure and L_2 regularization on the weights. The total loss for one training batch thus becomes:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{batch}} + \lambda \sum_{j=1}^{n_{\max}} w_j^2. \tag{3.16}$$

In this, we empirically set parameter λ to 0.001, and $\mathcal{L}_{\text{batch}}$ is either equal to the contrastive or median-based loss depending on the experiment. The batch size is equal to 100 text couples, of which 50 are related pairs and 50 are non-related pairs. An initial learning rate η of 0.01 is employed, which we immediately lower to 0.001 once the average epoch loss starts to deteriorate. After that, we stop training when the loss difference between epochs is below 0.05%. The weights are initialized uniformly to a value of 0.5. The entire procedure is visualised in Algorithm 3.1. To determine the optimal value of the hyperparameter κ in the median-based loss, we use five-fold cross-validation and a grid search procedure.

3.5.3 Results on Wikipedia

We train weights for Wikipedia texts with a fixed length of 20 words using the training procedure described in the previous section. The weights already converge before the first training epoch is finished, that is, without having seen all examples in the train set. This is due to the simplicity of our model—i.e. there are only limited degrees of freedom to be learned—and the large train set. Through cross-validation and grid-search we find

[3] deeplearning.net/software/theano

Algorithm 3.1: Detailed training procedure

$\forall j\colon w_j \leftarrow 0.5$
$\eta \leftarrow 0.01$
$\mathcal{L}_{\text{mean-old}} \leftarrow \infty$
repeat
  $\mathcal{L}_{\text{mean}} = 0$ `// new epoch`
  for *batch* $i \in$ *dataset* **do**
    $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{batch}_i} + \lambda \sum_{j=1}^{n_{\max}} w_j^2$
    $\forall j\colon w_j \leftarrow w_j - \eta \cdot \frac{\partial}{\partial w_j} \mathcal{L}_{\text{total}}$ `// gradient descent`
    $\mathcal{L}_{\text{mean}} \leftarrow \frac{(i-1)\mathcal{L}_{\text{mean}} + \mathcal{L}_{\text{total}}}{i}$ `// update mean loss`
  if $\eta > 0.001 \wedge \mathcal{L}_{mean} > \mathcal{L}_{mean\text{-}old}$ **then**
    $\eta \leftarrow 0.001$
  if $\eta == 0.001 \wedge \mathcal{L}_{mean\text{-}old} - \mathcal{L}_{mean} < 0.0005$ **then**
    STOP
  $\mathcal{L}_{\text{mean-old}} \leftarrow \mathcal{L}_{\text{mean}}$
until *STOP*

an optimal value for $\kappa = 160$ used in the median-based loss. The resulting weights for texts of 20 words long are visualized in Figure 3.3, for both the contrastive- and median-based loss. In both cases, the weights drop in magnitude with increasing index; this confirms the hypothesis that words with a low idf contribute less to the overall meaning of a sentence than high idf words. For the median-based loss, the first word is clearly the most important one, as the second weight is only half the first weight. It is important to point out that we observe a similar monotonically decreasing pattern for texts of 10 words and 30 words long, which means that we use an equal proportion of important to less important words, no matter how long the sentence is. From the 16th word on, the weights are close to zero, so these words almost never contribute to the meaning of a text. In comparison, there are, by experiment, eight non-informative words on average in a text of twenty words long.

The results of the experiments are summarized in Table 3.1. In a first experiment we compare the performance of our approach to all baselines for texts of length 20 and with word embeddings trained on Wikipedia. Our method significantly outperforms all baselines ($p < 0.001$, two-tailed binomial test) by a high margin for both losses. We also see that a plain

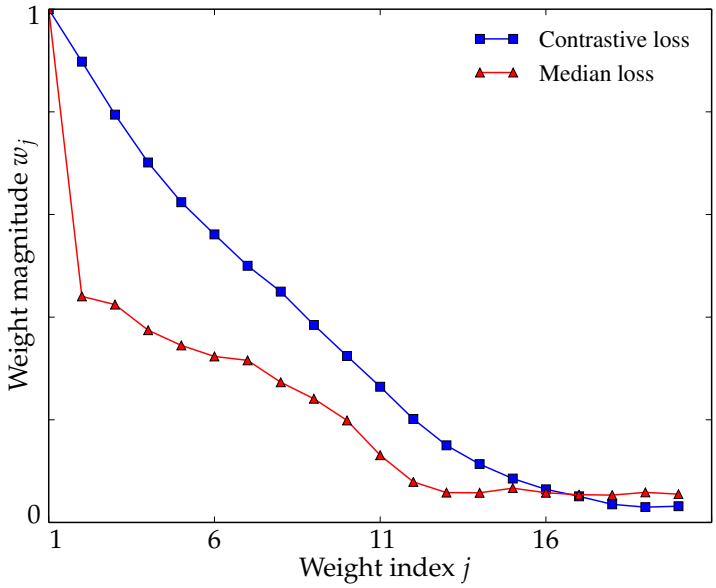


Figure 3.3: Plot of the learned weight magnitudes for texts of 20 words long.

tf-idf vector can compete with the simplest and most widely used baselines. We further notice that concatenating the minimum and maximum vectors performs approx 2.25% better than when using a maximum vector alone, which implies that the sign in word embeddings holds semantically relevant information.

In a second experiment we vary the length of the texts between 10 and 30 words. The overall performance drop can be addressed to the presence of text pairs with a length shorter than 20 words. Tf-idf now outperforms all baselines. For texts of 30 words the probability of word overlap is after all much higher than for texts of 10 words long; pairs of long texts thus help lower the error rate for tf-idf. Our method is still the overall best performer ($p < 0.001$, two-tailed binomial test), but this time the median-based loss is able to improve the contrastive-based loss by a high margin of almost 5% in split error, showing that the former loss is much more robust in a variable-length setting.

Finally, we also performed experiments with word embeddings from Google News, see footnote 1. We want to stress that we did not retrain the weights for this experiment in order to test whether our technique is indeed applicable out-of-the-box. There is only 20.6% vocabulary overlap between the Wikipedia and Google word2vec model, and there are only 300 dimensions instead of 400, which together can explain the overall per-

Table 3.2: Results for the Twitter data using Wikipedia embeddings.

	Split error	JS divergence
Tf-idf	43.09%	0.0634
Mean	33.68%	0.0783
Max	34.85%	0.0668
Min/max	33.80%	0.0734
Mean, top 30% idf	32.60%	0.0811
Max, top 30% idf	33.38%	0.0740
Min/max, top 30% idf	32.86%	0.0762
Mean, idf-weighted	31.28%	0.0886
Learned weights, contrastive	35.48%	0.0658
Learned weights, median	**30.88%**	**0.0900**

formance drop of the word embedding techniques. It is also possible that the Google embeddings are no right fit to be used on Wikipedia texts. Although our model was not trained to perform well on Google embeddings, we still achieve the best error rate of all embedding baselines ($p < 0.001$, two-tailed binomial test), and again the median loss outperforms the contrastive loss by approx 5%. Tf-idf, on the other hand, is the clear winner here; it did almost not suffer from the vocabulary reduction. It shows that vector dimensionality and context of usage are important factors in choosing or training word embeddings.

3.5.4 Results on Twitter

Next we perform experiments on the data collected from Twitter. We do not train additional word embeddings for Twitter, but we keep using the Wikipedia embeddings, since we have restricted ourselves to tweets of news publishers, who mainly use clean language. We also keep the same setting for κ as in the Wikipedia experiments.

The results for the Twitter experiments are shown in Table 3.2. As expected, the error rate is quite high given the noise present in the dataset. We also notice that the split error remains slightly higher than the human error rate of 28%. Tf-idf performs worst in this experiment. Compared to Wikipedia, tf-idf vectors for tweets are much sparser, which leads to a higher error rate. Tf-idf is clearly not fit to represent tweets efficiently. The baselines on the other hand have a much better, but overall comparable performance. Our method with median-based loss performs the best. The approach using contrastive loss performs worst among all embedding baselines, as during training the distribution of distances between related and between non-related texts rapidly gets skewed and develops addi-

tional modes. This causes the overall training loss to decrease, while the overlap between the related pairs' and non-related pairs' distribution further increases. The overall improvement of the median-based loss over the idf-weighted baseline is not statistically significant ($p > 0.05$, two-tailed binomial test); so, based on this Twitter dataset alone, we cannot draw any statistically sound conclusion whether our method is better in terms of split error than the idf-weighted baseline. Combined with the results from Table 3.1, however, we can conclude that choosing median-based learned weights is generally recommended.

3.6 Conclusion

We devised an effective method to derive vector representations for very short fragments of text. For this purpose we learned to weigh word embeddings based on their idf value, using both a contrastive-based loss function and a novel median-based loss function that can effectively mitigate the effect of outliers. Our method is applicable to texts of a fixed length, but can easily be extended to texts of a variable length through subsampling and linear interpolation of the learned weights. Our method can be applied out-of-the-box, that is, there is no need to retrain the model when using different types of word embeddings. We showed that our method outperforms widely-used baselines that naively combine word embeddings into a text representation, using both toy Wikipedia and real-word Twitter data. All code for this paper is readily available on our GitHub page github.com/cedricdeboom/RepresentationLearning.

References

[1] C. De Boom, S. Van Canneyt, and B. Dhoedt. *Semantics-driven Event Clustering in Twitter Feeds.* In #MSM, 2015.

[2] N. Jonnalagedda and S. Gauch. *Personalized News Recommendation Using Twitter*. In Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on, 2013.

[3] P. Achananuparp, X. Hu, and X. Shen. *The Evaluation of Sentence Similarity Measures*. In DaWaK 2008: International Conference on Data Warehousing and Knowledge Discovery, 2008.

[4] C. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. *Indexing by Latent Semantic Analysis.* JASIS, 1990.

[6] D. M. Blei, A. Y. Ng, and M. I. Jordan. *Latent dirichlet allocation*. Machine Learning, 2003.

[7] T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. In Proceedings of Workshop at ICLR, 2013.

[8] R. Lebret and R. Collobert. *N-gram-Based Low-Dimensional Representation for Document Classification.* arXiv.org, 2015. arXiv:1412.6277.

[9] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. *Natural Language Processing (Almost) from Scratch*. The Journal of Machine Learning Research, 2011.

[10] B. Hu, Z. Lu, H. Li, and Q. Chen. *Convolutional Neural Network Architectures for Matching Natural Language Sentences*. In NIPS 2014: Advances in Neural Information Processing Systems, 2014.

[11] J. Xu, P. Wang, G. Tian, B. Xu, J. Zhao, and F. Wang. *Short Text Clustering via Convolutional Neural Networks*. In Proceedings of NAACL- . . . , 2015.

[12] R. Johnson and T. Zhang. *Semi-Supervised Learning with Multi-View Embedding: Theory and Application with Convolutional Neural Networks*. arXiv.org, 2015. arXiv:1504.01255v1.

[13] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. *A Convolutional Neural Network for Modelling Sentences*. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, 2014.

[14] I. Sutskever, O. Vinyals, and Q. V. Le. *Sequence to Sequence Learning with Neural Networks*. In NIPS 2014, 2014.

[15] A. Sordoni, Y. Bengio, H. Vahabi, C. Lioma, J. G. Simonsen, and J.-Y. Nie. *A Hierarchical Recurrent Encoder-Decoder For Generative Context-Aware Query Suggestion*. arXiv.org, 2015. arXiv:1507.02221v1.

[16] M. Sundermeyer, H. Ney, and R. Schluter. *From Feedforward to Recurrent LSTM Neural Networks for Language Modeling*. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 2015.

[17] R. Kiros, Y. Zhu, R. Salakhutdinov, R. S. Zemel, A. Torralba, R. Urtasun, and S. Fidler. *Skip-Thought Vectors*. arXiv.org, 2015. arXiv:1506.06726v1.

[18] Q. V. Le and T. Mikolov. *Distributed Representations of Sentences and Documents*. arXiv.org, 2014. arXiv:1405.4053v2.

[19] J. Weston, S. Chopra, and K. Adams. *#TagSpace: Semantic embeddings from hashtags*. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014.

[20] C. N. dos Santos and M. Gatti. *Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts*. In COLING 2014, the 25th International Conference on Computational Linguistics, 2014.

[21] W. Yin and H. Schütze. *Convolutional Neural Network for Paraphrase Identification.* In HLT-NAACL, 2015.

[22] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger. *From Word Embeddings To Document Distances.* ICML, 2015.

[23] T. Kenter and M. de Rijke. *Short Text Similarity with Word Embeddings*. In International Conference on Information and Knowledge Management, 2015.

[24] L. Kang, B. Hu, X. Wu, Q. Chen, and Y. He. *A Short Texts Matching Method using Shallow Features and Deep Features*. In Third CCF Conference, NLPCC 2014, 2014.

[25] J. Bromley, I. Guyon, Y. Lecun, E. Säckinger, and R. Shah. *Signature Verification Using a Siamese Time Delay Neural Network.* NIPS, 1993.

[26] R. Hadsell, S. Chopra, and Y. Lecun. *Dimensionality Reduction by Learning an Invariant Mapping*. In CVPR, 2006.

[27] C. De Boom, S. Van Canneyt, S. Bohez, T. Demeester, and B. Dhoedt. *Learning Semantic Similarity for Very Short Texts*. In International Conference on Data Mining Workshop, 2015.

4

Character-level Recurrent Neural Networks in Practice: Comparing Training and Sampling Schemes

"Alles is, alles is, alles is belangrijk. Zo voorbij, zo voorbij. Wie heeft er tijd, want het is zo voorbij."

—Buurman, 2017

The main issue with the text representation discussed in Chapter 3 is that the order of the words is not preserved. In this chapter, we will therefore explore truly sequential models for text representations, which process one token after the other in a sequence. More specifically, we will focus on recurrent neural networks that take individual characters as input, so-called 'character-level recurrent neural networks'. The goal of this chapter is to provide the user of such models with some practical guidelines and details on the training and sampling procedures. In chapters 5 and 6 we will use recurrent neural networks in some practical applications, such as music generation and recommender systems.

⋆ ⋆ ⋆

C. De Boom, T. Demeester, and B. Dhoedt.

Appeared in Neural Computing and Applications, online, 2018.

Abstract Recurrent neural networks are nowadays successfully used in an abundance of applications, going from text, speech and image processing to recommender systems. Backpropagation through time is the algorithm that is commonly used to train these networks on specific tasks. Many deep learning frameworks have their own implementation of training and sampling procedures for recurrent neural networks, while there are in fact multiple other possibilities to choose from and other parameters to tune. In existing literature this is very often overlooked or ignored. In this paper we therefore give an overview of possible training and sampling schemes for character-level recurrent neural networks to solve the task of predicting the next token in a given sequence. We test these different schemes on a variety of datasets, neural network architectures and parameter settings, and formulate a number of take-home recommendations. The choice of training and sampling scheme turns out to be subject to a number of trade-offs, such as training stability, sampling time, model performance and implementation effort, but is largely independent of the data. Perhaps the most surprising result is that transferring hidden states for correctly initializing the model on subsequences often leads to unstable training behavior depending on the dataset.

4.1 Introduction

Dynamic sequences of discrete tokens are abundant and we encounter them on a daily basis. Examples of discrete tokens are characters or words in a text, notes in a musical composition, pixels in an image, actions in a reinforcement learning agent, web pages one visits, tracks one listens to on a music streaming service etc. Each of these tokens appears in a sequence, in which there is often a strong correlation between consecutive or nearby tokens. For example, the similarity between neighboring pixels in an image is very large since they often share similar shades of colors. Words in sentences, or characters in words, are also correlated because of the underlying semantics and language characteristics.

In this paper only discrete tokens are considered as opposed to sequences of real-valued samples, such as stock prices, analog audio signals, word embeddings, etc. but our methodology is also applicable to these kinds of sequences. A sequence of discrete tokens can be presented to a machine learning model that is designed to assess the probability of the next token in the sequence by modeling $p(x_n|x_{n-1}, x_{n-2}, \ldots, x_1)$, in which x_i is the i'th token in the sequence. These kinds of models are the underlying mechanisms of autoregressive models [1], recurrent and recursive models, dynamical systems etc. In the field of natural language processing (NLP)

they are called language models, in which each token stands for a separate word or n-gram [2]. Since these models give us a probability distribution of the next token in the sequence, a sample from this distribution can be drawn and thus a new token for the sequence is generated. By recursively applying this generation step, entire new sequences can be generated. In NLP, for example, a language model is not only capable of assessing proper language utterances but also of generating new and unseen text.

One particular type of generative models that has become popular in the past years is the recurrent neural network (RNN). In a regular neural network a fixed-dimensional feature representation is transformed into another feature representation through a non-linear function; multiple instances of such feature transformations are applied to calculate the final output of the neural network. In a recurrent neural network this process of feature transformations is also repeated in time: at every time step a new input is processed and an output is produced, which makes it suitable for modeling time series, language utterances etc. These dynamic sequences can be of variable length, and RNNs are able to effectively model semantically rich representations of these sequences. For example, in 2013, Graves showed that RNNs are capable of generating structured text, such as a Wikipedia article, and even continuous handwriting [3]. From then on these models have shown great potential at modeling the temporal dynamics of text, speech as well as audio signals [4–6]. Recurrent neural networks can also effectively generate new images on a per pixel basis, as was shown by Gregor et al. with DRAW [7] and by van den Oord et al. with PixelRNNs [8]. Next to this, in the context of recommender systems, RNNs have been used successfully to model user behavior on online services and to recommend new items to consume [9, 10].

Despite the fact that RNNs are abundant in scientific literature and industry, there is not much consensus on how to efficiently train these kinds of models, and, to the extent of our knowledge, there are no focused contributions in literature that tackle this question. The choice of training algorithm very often depends on the deep learning framework at hand, while in fact there are multiple factors that influence the RNN performance, and those are often ignored or overlooked. Merity et al. have pointed out before that "[the] training of RNN models [...] has fundamental trade-offs that are rarely discussed" [11]. The goal of this paper is to study a number of widely applicable training and sampling techniques for RNNs, along with their respective (dis)advantages and trade-offs. These will be tested on a variety of datasets, neural network architectures, and parameter settings, in order to gain insights into which algorithm is best suited. In the next section the concept of RNNs is introduced and, more specifically, character-

level RNNs, and how these models are trained. In Section 4.3 four different training and sampling methods for RNNs are detailed. After that, in Section 4.4, we will present experimental results on the accuracy, efficiency and performance of the different methods. We will also present a set of take-home recommendations and a range of future research tracks. Finally, the conclusions are listed in Section 4.6. Table 4.1 gives an overview of the symbols that will be used throughout this paper in order of appearance.

Table 4.1: Tabel of symbols in order of appearance.

x_i, $\mathbf{x}_i$	Input token i, input vector i
h_i, $\mathbf{h}_i$	Hidden state i, hidden state vector i
y_i, $\mathbf{y}_i$	Output token i, output vector i
$f(\cdot), g(\cdot)$	Parameterized and differentiable functions
$\mathcal{L}(\cdot)$	Loss function
η	Learning rate
w, $\mathbf{W}$	Arbitrary weight, arbitrary weight matrix
k_1	Number of time steps after which one truncated BPTT operation is performed
k_2	Number of time steps over which gradients are backpropagated in truncated BPTT
$\phi(\cdot)$	Nonlinear activation function
$\sigma(\cdot)$	Sigmoid function: $\sigma(x) = 1/(1+\exp(-x))$
$\mathcal{R}$	RNN model
$\mathcal{R}(\cdot)$	Output function of RNN model $\mathcal{R}$
$\mathcal{R}_i$	Hidden state i of RNN model $\mathcal{R}$
$\odot$	Element-wise vector multiplication operator
$\oplus$	Sequence concatenation operator
$\mathcal{D}$, $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{test}}$	Dataset, train set, test set
V	Ordered set of tokens appearing in a dataset ('vocabulary')
r	Number of recurrent layers in an RNN model
γ	Dimensionality of the recurrent layers in an RNN model

4.2 Character-level Recurrent Neural Networks

As mentioned in the introduction, this paper mainly focuses on dynamic sequences of discrete tokens. Generating and modeling such sequences is the core application of a specific type of recurrent neural networks: character-level recurrent neural networks. Recurrent neural networks (RNN) are designed to maintain a summary of the past sequence in their memory or so-called *hidden state*, which is updated whenever a new input token is presented. This summary is used to make a prediction about the next token in the sequence, i.e. the model $p(x_n|h_{n-1})$ in

which the hidden state h_{n-1} of the RNN is a function of the past sequence $(x_{n-1}, x_{n-2}, \ldots, x_1)$. Formally, we have:

$$\begin{aligned} h_n &= f(x_n, h_{n-1}), \\ x_{n+1} &= g(h_n). \end{aligned} \tag{4.1}$$

Given an adequate initialization h_0 of the hidden state and trained parameterized functions $f(\cdot)$ and $g(\cdot)$, the previous scheme can be used to generate an infinite sequence of tokens. In character-level RNNs specifically, all input tokens are discrete and $g(\cdot)$ is a stochastic function that produces a probability mass function over all possible tokens. To produce the next token in the sequence, one can sample from this mass function or simply pick the token with the highest probability:

$$\begin{aligned} h_n &= f(x_n, h_{n-1}), \\ x_{n+1} &\sim g(h_n). \end{aligned} \tag{4.2}$$

Since the hidden state at time step n is only dependent on the tokens up to time n and not on future tokens, a character-level RNN can be regarded as the following fully probabilistic generative model [12]:

$$p(x_{1:N}) = \prod_{n=1}^{N} p(x_n | x_{1:n-1}) = \prod_{n=1}^{N} p(x_n, h_{n-1}). \tag{4.3}$$

In this, we have used the slice notation $x_{k:\ell}$, which means $(x_k, x_{k+1}, \ldots, x_\ell)$. As a side comment, even though their name only refers to character-based language models, character-level RNNs are fit to model a wide variety of discrete sequences, for which we refer the reader to the introduction section.

4.2.1 Truncated backpropagation through time

Regular feedforward neural networks are trained using the *backpropagation* algorithm [13]. In this, a certain input is first propagated through the network to compute the output. This is called the forward pass. The output is then compared to a ground truth label using a differentiable loss function. In the backward pass the gradients of the loss with respect to all the parameters in the network are computed by application of the chain rule. Finally, all parameters are updated using a gradient-based optimization procedure such as gradient descent [14]. In neural network terminology, the parameters of the network are also called the weights. If the loss function between the network output y and the ground truth label $\hat{y}$ is denoted by $\mathcal{L}(y, \hat{y})$,

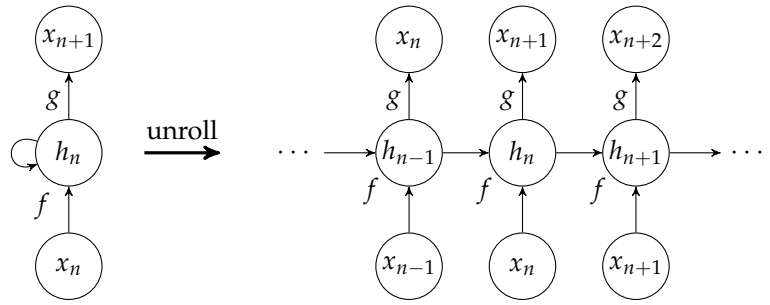


Figure 4.1: Unrolling a recurrent neural network in time. Functions $f(\cdot)$ and $g(\cdot)$ and their parameters are shared across all time steps.

and the vector of all weights in the network by $\mathbf{w}$, a standard update rule in gradient descent is given by:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}(y, \hat{y}). \tag{4.4}$$

Here, η is the so-called learning rate, which controls the size of the steps taken with each weight update. In practice, input samples will be organized in equally-sized batches sampled from the training dataset for which the loss is averaged or summed, leading to less noisy updates. This is called mini-batch gradient descent. Other gradient descent flavors such as RMSprop and Adam further extend on Equation (4.4), making the optimization procedure even more robust [15].

In recurrent neural networks a new input is applied for every time step, and the output at a certain time step is dependent on all previous inputs, as was shown in Equation (4.3). This means that the loss at time step N needs to be backpropagated up until the applied inputs at time step 1. This procedure is therefore called *backpropagation through time* (BPTT) [12]. If the sequence is very long, BPTT quickly becomes inefficient: backpropagating through 100 time steps can be compared to backpropagating through a 100-layer deep feedforward neural network. Unlike with feedforward networks however, in RNNs the weights are shared across time. This can best be seen if we unroll the RNN to visualize the separate time steps, see Figure 4.1.

To scale backpropagation through time for use with long sequences, the gradient update is often halted after having traversed a fixed number of time steps. Such a procedure is called *truncated backpropagation through time*. Apart from stopping the gradient updates from backpropagating all the way to the beginning of the sequence, we also limit the frequency of such updates. For a given training sequence, truncated BPTT then proceeds as follows. Every time step a new token is processed by the RNN,

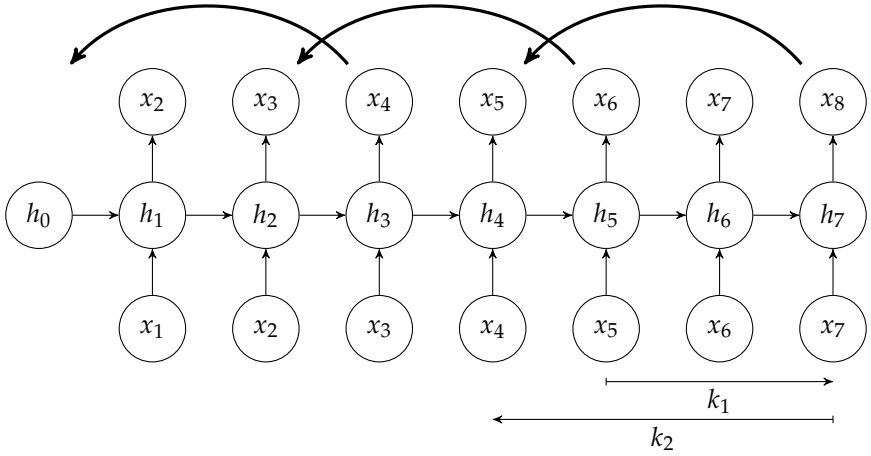


Figure 4.2: Example of trunctated backpropagation through time for $k_1 = 2$ and $k_2 = 3$. The thick arrows indicate one backpropagation through time update.

and whenever k_1 tokens have been processed in the so-called *forward pass*—and the hidden state is updated k_1 times—truncated BPTT is initiated by backpropagating the gradients for k_2 time steps. Here, by analogy with Sutskever [12], we have denoted the number of time steps between performing truncated BPTT by k_1 and the length of the BPTT by k_2. We will keep using these parameters throughout this paper. A visual explanatory example of truncated BPPT can be found in Figure 4.2, which shows how every two (k_1) time steps the gradients are backpropagated for three (k_2) time steps. Note that, in order to remain as data efficient as possible, k_1 should preferably be less than or equal to k_2, since otherwise some data points would be skipped during training.

4.2.2 Common RNN layers

As mentioned before, RNNs keep a summary of the past sequence encoded in a hidden state representation. Whenever a new input is presented to the RNN, this hidden state gets updated. The way in which the update happens depends on the internal dynamics of the RNN. The simplest version of an RNN is an extension of a feedforward neural network, in which matrix multiplications are used to perform input transformations:

$$\mathbf{u}_{i+1} = \phi\left(\mathbf{W}_i \mathbf{u}_i + \mathbf{b}_i\right) \tag{4.5}$$

in which $\mathbf{u}_i$ is the vector representation at the i'th layer of the neural network, $\mathbf{W}_i$ is the matrix containing the weights of this layer, and $\mathbf{b}_i$ is the vector of biases. The function $\phi(\cdot)$ is a nonlinear transformation function, also called activation function; often used examples are the sigmoid logistic function $\sigma(\cdot)$, $\tanh(\cdot)$ or rectified linear units (ReLU) and their variants. In the case of RNNs, the hidden state update is rewritten in the style of Equation (4.1) as follows:

$$\mathbf{h}_n = \phi\left(\mathbf{W}_\mathbf{x}\mathbf{x}_n + \mathbf{W}_\mathbf{h}\mathbf{h}_{n-1} + \mathbf{b}\right). \tag{4.6}$$

The output of the RNN can be computed with Equation (4.5) using $\mathbf{h}_n$ as input vector.

To help the RNN model long-term dependencies and to counter the vanishing gradient problem [16], several extensions on Equation (4.6) have been proposed. The best known examples are long short-term memories (LSTMs) and, more recently, gated recurrent units (GRUs), which both have comparable characteristics and similar performance on a variety of tasks [17–19]. Both LSTMs and GRUs incorporate a gating mechanism which controls to what extent the new input is stored in memory and the old memory is forgotten. For this purpose the LSTM introduces an input ($\mathbf{i}_n$) and forget ($\mathbf{f}_n$) gate, as well as an output gate ($\mathbf{o}_n$):

$$\begin{aligned}
\mathbf{i}_n &= \sigma\left(\mathbf{U}_\mathbf{i}\mathbf{x}_n + \mathbf{W}_\mathbf{i}\mathbf{h}_{n-1} + \mathbf{w}_\mathbf{i} \odot \mathbf{c}_{n-1} + \mathbf{b}_\mathbf{i}\right), \\
\mathbf{f}_n &= \sigma\left(\mathbf{U}_\mathbf{f}\mathbf{x}_n + \mathbf{W}_\mathbf{f}\mathbf{h}_{n-1} + \mathbf{w}_\mathbf{f} \odot \mathbf{c}_{n-1} + \mathbf{b}_\mathbf{f}\right), \\
\mathbf{c}_n &= \mathbf{f}_t \odot \mathbf{c}_{n-1} + \mathbf{i}_n \odot \tanh\left(\mathbf{U}_\mathbf{c}\mathbf{x}_n + \mathbf{W}_\mathbf{c}\mathbf{h}_{n-1} + \mathbf{b}_\mathbf{c}\right), \\
\mathbf{o}_n &= \sigma\left(\mathbf{U}_\mathbf{o}\mathbf{x}_n + \mathbf{W}_\mathbf{o}\mathbf{h}_{n-1} + \mathbf{w}_\mathbf{o} \odot \mathbf{c}_{n-1} + \mathbf{b}_\mathbf{o}\right), \\
\mathbf{h}_n &= \mathbf{o}_n \odot \tanh\left(\mathbf{c}_n\right),
\end{aligned} \tag{4.7}$$

Here, the symbol $\odot$ stands for the element-wise vector multiplication. Note that the LSTM uses two types of memories: a hidden state $\mathbf{h}_n$ and a so-called cell state $\mathbf{c}_n$. Compared to LSTMs, GRUs do not have this extra cell state and only incorporate two gates: a reset and update gate [20]. This reduces the overall number of parameters and generally speeds up learning. The choice of LSTMs versus GRUs is dependent on the application at hand, but in the experiments of this paper we will use LSTMs as these are currently the most widely used RNN layers.

Since in neural networks multiple layers are usually stacked onto one another, this is also possible with recurrent layers. In that case, the output at time step n is fed to the input of the next recurrent layer, also at time step n. Each layer thus processes the sequence of outputs produced by the previous layer. This will, of course, significantly slow down BPTT.

4.3 Training and sampling schemes for character-level RNNs

In this section four schemes are presented on how to train character-level RNNs and how to sample new tokens from them. The task of the RNN model is independent of these schemes and its purpose is to predict the next symbol or character in a sequence given all previous tokens. The training and sampling schemes are thus merely a practical means to solve the same task, and in later sections the effect of the used scheme on the performance and efficiency of the RNN model is studied. We already point out that the four schemes presented are among the most basic and practical methods to train and sample from RNNs, but of course many more combinations or variants could be devised. In the discussion of the different schemes a general character-level RNN will be denoted by $\mathcal{R}$, and $\mathcal{R}(x)$ is the output of the RNN by applying token x at its input. This output is a vector that represents a probability distribution across all characters. For notational convenience, we write the i'th hidden state of the RNN by $\mathcal{R}_i$.

4.3.1 High-level overview

As mentioned before, we will isolate four different schemes on how to train RNNs and how to sample new tokens from them. Each scheme fits in the truncated BPTT framework, and is in fact a practical approximation of the original algorithm. So whenever we use the k_1 and k_2 parameters, these refer to the definitions we gave in Section 4.2.1. It is also important to keep in mind that the task for all schemes is the same, namely to predict the next token in a given sequence.

To help understand the mechanisms of each scheme, we visualized them schematically in Figures 4.3, 4.4, 4.5 and 4.6. In the training procedures we have drawn the output tokens for which a loss is defined. We see that, for example, the main difference of scheme 2 compared to scheme 1 is that we only compute a loss for the final output token instead of for all output tokens during training. Regarding the sampling procedures, in the first two schemes a new token is always sampled starting from the same initial hidden state, which is colored light gray. We call this principle 'windowed sampling'. In schemes 3 and 4, on the other hand, sampling a new token is based on the current hidden state and by applying the previous token at the input of the RNN. This sampling procedure is called 'progressive sampling'. In the training procedure of scheme 4 we observe a similar technique, in which the hidden state is carried across subsequent sequences. In the next subsections we will give details on all training and

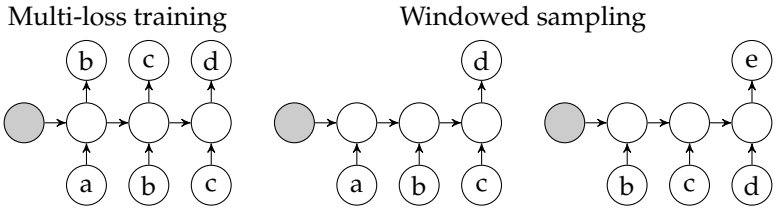


Figure 4.3: Graphical visualization of scheme 1.

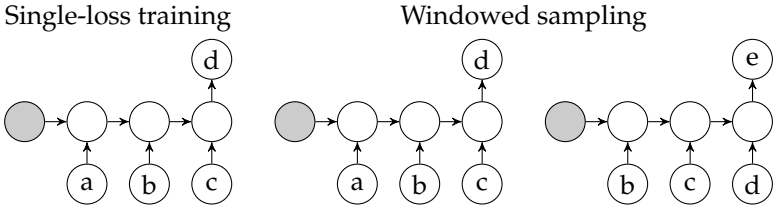


Figure 4.4: Graphical visualization of scheme 2.

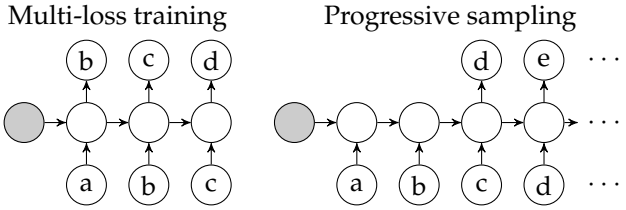


Figure 4.5: Graphical visualization of scheme 3.

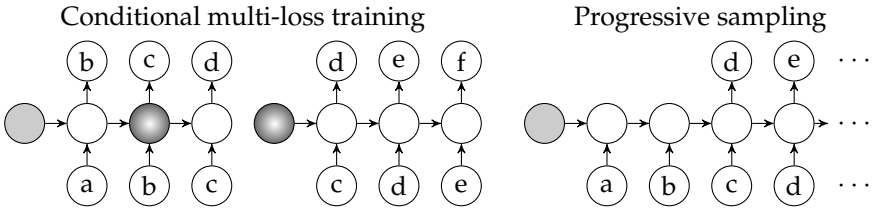


Figure 4.6: Graphical visualization of scheme 4. The shaded circle is the remembered hidden state.

sampling procedures, after which we go over the practical details of the different schemes one by one. We mention that the schemes are described without batching, while in practice mini-batch training is usually done, as motivated in Section 4.2.1. The schemes, however, are easily transferred to a batched setting.

4.3.2 Training algorithms

We have isolated three different training procedures for character-level RNNs. A first algorithm is called **multi-loss training**, and a rudimentary outline of this is shown in Algorithm 4.1. The input sequences all have length k_2 and are subsequently taken from the train set by skipping every k_1 characters. For each input token a loss is calculated at the output of the RNN. When the entire sequence is processed, the average of all losses which we use to update the RNN weights is calculated. We also reset the hidden state of the RNN for each new training sequence. This initial hidden state will be learned through backpropagation together with the weights of the RNN. In practice, for every input sequence of k_2 characters, the initial hidden state will be the same. Note that for LSTMs, the hidden state comprises both the hidden and cell vectors from Equation (4.7). The `truncated_BPTT` procedures on lines 10 and 12 calculates the gradients of the loss with respect to all weights in the RNN using backpropagation. The `optimize` procedure on lines 11 and 13 then uses these gradients to update the weights using (a variant of) Equation (4.4).

Algorithm 4.1: Multi-loss training procedure

input : dataset $\mathcal{D}$ of tokens, RNN $\mathcal{R}$, initial hidden state vector $\mathbf{h}_0$, loss function $\mathcal{L}$

parameters: truncated BPTT parameters k_1 and k_2, learning rate η

1 **repeat**
2 $\quad j \leftarrow 1$
3 $\quad$ **while** $j < size(\mathcal{D}) - k_2$ **do**
4 $\quad\quad \mathsf{loss} \leftarrow 0$
5 $\quad\quad \mathcal{R}_0 \leftarrow \mathbf{h}_0$
6 $\quad\quad s \leftarrow \mathcal{D}_{j:j+k_2}$
7 $\quad\quad$ **foreach** *token pair* $(x_i, x_{i+1}) \in s$ **do**
8 $\quad\quad\quad \mathbf{y} \leftarrow \mathcal{R}(x_i)$
9 $\quad\quad\quad \mathsf{loss} \leftarrow \mathsf{loss} + \mathcal{L}(\mathbf{y}, x_{i+1})/k_2$
10 $\quad\quad \forall w \in \mathcal{R} : {}^{\partial \mathsf{loss}}/_{\partial w} \leftarrow \texttt{truncated\_BPTT}(\mathsf{loss}, w, k_2)$
11 $\quad\quad \forall w \in \mathcal{R} : \texttt{optimize}(w, {}^{\partial \mathsf{loss}}/_{\partial w}, \eta)$
12 $\quad\quad \nabla_{\mathbf{h}_0}\mathsf{loss} \leftarrow \texttt{truncated\_BPTT}(\mathsf{loss}, \mathbf{h}_0, k_2)$
13 $\quad\quad \texttt{optimize}(\mathbf{h}_0, \nabla_{\mathbf{h}_0}\mathsf{loss}, \eta)$
14 $\quad\quad j \leftarrow j + k_1$
15 **until** *convergence*

In **single-loss training**, instead of defining a loss on all outputs of the RNN—which forces the RNN to make good predictions for the first few tokens of the sequence—we only define a loss on the final predicted token in the sequence. The complete training procedure is shown in Algorithm 4.2. The difference with Algorithm 4.1 is that the inner most loop does not aggregate the loss for every RNN output. Now the loss is calculated outside this loop on line 9, only for the final RNN output.

In both the multi-loss and single-loss procedures we always start training on a sequence from an initial hidden state that is learned. In **conditional multi-loss training**, on the other hand, the multi-loss training procedure is adapted to reuse the hidden state across different sequences. Such an approach leans much closer to the original truncated BPTT algorithm than when the initial state is always reset. The outline of the training method is given in Algorithm 4.3. Since we are using truncated BPTT, the procedure requires meticulous bookkeeping of the hidden state at every time step, which can be observed in lines 11–12. This is especially true when we work in a mini-batch setting where we also need to keep track of how the subsequent batches are constructed.

Algorithm 4.2: Single-loss training procedure

input : dataset $\mathcal{D}$ of tokens, RNN $\mathcal{R}$, initial hidden state vector $\mathbf{h}_0$, loss function $\mathcal{L}$

parameters: truncated BPTT parameters k_1 and k_2, learning rate η

1 **repeat**
2 $\quad j \leftarrow 1$
3 $\quad$ **while** $j < size(\mathcal{D}) - k_2$ **do**
4 $\quad\quad \mathcal{R}_0 \leftarrow \mathbf{h}_0$
5 $\quad\quad s \leftarrow \mathcal{D}_{j:j+k_2}$
6 $\quad\quad$ **foreach** *token pair* $(x_i, x_{i+1}) \in s$ **do**
7 $\quad\quad\quad \mathbf{y} \leftarrow \mathcal{R}(x_i)$
8 $\quad\quad\quad y' \leftarrow x_{i+1}$
9 $\quad\quad \mathsf{loss} \leftarrow \mathcal{L}(\mathbf{y}, y')$
10 $\quad\quad \forall w \in \mathcal{R} : {}^{\partial \mathsf{loss}}/_{\partial w} \leftarrow \texttt{truncated\_BPTT}(\mathsf{loss}, w, k_2)$
11 $\quad\quad \forall w \in \mathcal{R} : \texttt{optimize}(w, {}^{\partial \mathsf{loss}}/_{\partial w}, \eta)$
12 $\quad\quad \nabla_{\mathbf{h}_0}\mathsf{loss} \leftarrow \texttt{truncated\_BPTT}(\mathsf{loss}, \mathbf{h}_0, k_2)$
13 $\quad\quad \texttt{optimize}(\mathbf{h}_0, \nabla_{\mathbf{h}_0}\mathsf{loss}, \eta)$
14 $\quad\quad j \leftarrow j + k_1$
15 **until** *convergence*

Algorithm 4.3: Conditional multi-loss training procedure

input : dataset $\mathcal{D}$ of tokens, RNN $\mathcal{R}$, loss function $\mathcal{L}$
parameters: truncated BPTT parameters k_1 and k_2, learning rate η

1 **repeat**
2 $j \leftarrow 0$
3 $\mathbf{h}_0 \leftarrow \mathbf{0}$
4 **while** $j < size(\mathcal{D}) - k_2$ **do**
5 $\mathsf{loss} \leftarrow 0$
6 $\mathcal{R}_0 \leftarrow \mathbf{h}_0$
7 $s \leftarrow \mathcal{D}_{j:j+k_2}$
8 **foreach** *token pair* $(x_i, x_{i+1}) \in s$ **do**
9 $\mathbf{y} \leftarrow \mathcal{R}(x_i)$
10 $\mathsf{loss} \leftarrow \mathsf{loss} + \mathcal{L}(\mathbf{y}, x_{i+1})/k_2$
11 **if** $i == k_1 - 1$ **then**
12 $\mathbf{h}_0 \leftarrow \mathcal{R}_i$
13 $\forall w \in \mathcal{R} : {}^{\partial \mathsf{loss}}/_{\partial w} \leftarrow \texttt{truncated\_BPTT}(\mathsf{loss}, w, k_2)$
14 $\forall w \in \mathcal{R} : \texttt{optimize}(w, {}^{\partial \mathsf{loss}}/_{\partial w}, \eta)$
15 $j \leftarrow j + k_1$
16 **until** *convergence*

4.3.3 Sampling algorithms

Next to the training algorithms we have explained in the previous section, we also discern two different sampling procedures. These are used to generate new and previously unseen sequences. Both procedures have in common that sampling is started with a seed sequence of k_2 tokens that is fed to the RNN. This is done in order to appropriately bootstrap the RNN's hidden state. After the seed sequence has been processed, the two procedures start to differ.

In so-called **windowed sampling** the next token of the sequence is sampled from the RNN after applying the seed sequence. This newly sampled token is concatenated at the end of the sequence. After this, the hidden state of the RNN is reset to its learned representation. Sampling the next token proceeds in the same way: we take the last k_2 tokens from the sequence that have been sampled thus far, we feed them to the RNN, the next token is sampled and appended to the sequence, and the hidden state of the RNN is reset. The entire windowed sampling procedure is sketched in Algorithm 4.4. On line 6 of the algorithm, we have used the symbol $\oplus$ to indicate sequence concatenation.

Algorithm 4.4: Windowed sampling procedure

input : RNN $\mathcal{R}$, initial hidden state vector $\mathbf{h}_0$, seed sequence s
parameters: truncated BPTT parameter k_2

```
1 repeat
2     R_0 ← h_0
3     s' ← get_last_k_tokens(s, k = k_2)
4     foreach token x ∈ s' do
5         y ← R(x)
6     s ← s ⊕ sample(y)
7 until enough tokens in s
```

Algorithm 4.5: Progressive sampling procedure

input : RNN $\mathcal{R}$, initial hidden state vector $\mathbf{h}_0$, seed sequence s
parameters: truncated BPTT parameter k_2

```
1 R_0 ← h_0
2 foreach token x ∈ s do
3     y ← R(x)
4 repeat
5     t ← sample(y)
6     s ← s ⊕ t
7     y ← R(t)
8 until enough tokens in s
```

In **progressive sampling** the next token in a sequence is always sampled given the current hidden state and the previously sampled token. That is, a token is applied at the input of the RNN, which updates its hidden state, and then the next token is sampled at the RNN output. The initial hidden state is therefore never reset. This is the most intuitive way of sampling from an RNN. The entire sampling procedure is given in Algorithm 4.5. On lines 1–3 the RNN is bootstrapped using the initial hidden state and the seed sequence. On the following lines, new tokens are continuously sampled from the RNN one token at a time, so the inner loop from Algorithm 4.4 is no longer needed.

4.3.4 Scheme 1 – Multi-loss training, windowed sampling

In a first scheme, multi-loss training (Algorithm 4.1) is combined with a windowed sampling procedure (Algorithm 4.4). The main advantage of using this scheme is that there is no need for hidden state bookkeeping across sequences. Especially during training this can be cumbersome in a batched version of the algorithm. One disadvantage is that sampling is slower if k_2 increases: to sample one new token k_2 inputs need to be processed. If the RNN model contains many layers, this can lead to scalability issues. Another disadvantage is that a loss is defined on all k_2 outputs of the RNN during training. That is, we force the RNN to produce good token candidates after having seen only one or a few input tokens. This can lead to a short-sighted RNN model that mostly looks at the more recent history to make a prediction for the next token. In scheme 2 this potential issue is solved using single-loss training.

4.3.5 Scheme 2 – Single-loss training, windowed sampling

In the second scheme, the multi-loss training procedure of scheme 1 is replaced by the single-loss equivalent of Algorithm 4.2. The main advantage is that we allow the hidden state of the RNN a certain burn-in period, so that predictions can be made using more knowledge from the past sequence. Burning in the hidden state also causes the RNN to be able to learn long-term dependencies in the data, because we only make a prediction after having seen k_2 tokens. The potential drawback is that learning is slower, since only one signal is backpropagated for every sequence compared to k_2 signals in the first scheme. The sampling algorithm, on the other hand, is the same as in the first scheme, and now almost perfectly reflects how the RNN has been trained, i.e. by only considering the final token for each input sequence.

4.3.6 Scheme 3 – Multi-loss training, progressive sampling

In scheme number 3, we go back to the multi-loss training procedure of scheme 1, but now the progressive sampling from Algorithm 4.5 is used instead of windowed sampling. One drawback of the sampling method in scheme 1 is that it is not very scalable for large values of k_2, since we need to feed a sequence of k_2 tokens to the RNN for every token that is sampled. In progressive sampling, on the other hand, the next token is sampled immediately for every new input token. This way, the sampling of new sequences is sped up by a factor of approximately k_2, which is the main advantage of this scheme.

4.3.7 Scheme 4 – Conditional multi-loss training, progressive sampling

In scheme 3 we still use standard multi-loss training, which resets the hidden state for every train sequence. Scheme 4 replaces this by the conditional multi-loss training procedure from Algorithm 4.3, while maintaining the progressive sampling algorithm. One of the main disadvantages of using this particular training algorithm, is its requirement to keep track of the hidden states across train sequences and to carefully select these train sequences from the dataset, which can be hard in mini-batch settings. Next to this, whenever the RNN weights are updated, the hidden state from before the update is reused, which can potentially lead to unstable learning behavior. On the plus side, we are able to learn dependencies between tokens that are more than k_2 time steps away, since the hidden state is remembered in between train sequences. Also, the need to learn an appropriate initial hidden state is eliminated, which can lead to a small speed-up in learning.

4.3.8 Literature overview

We will now go over some of the works in literature that have used RNNs for language modeling, on both character and word level. Most of the works that are listed, describe or have described state-of-the-art results on famous benchmarks such as the Penn Treebank dataset [40, 41], WikiText-2 [11] and the Hutter Prize datasets [42]. The first two datasets are mainly used to benchmark word-level language models, while the Hutter Prize datasets are generally used for character-level evaluation. Some papers, however, also train character-level models on the Penn Treebank dataset. It is our purpose to give the reader a high-level idea of what schemes are being used in existing literature. We do not intend to give a complete overview of the literature on RNNs for language modeling. Instead, we focus on highly cited works that have, at some point, reported state-of-the-art results on some of the above-mentioned benchmarks. In this, attention is given to the most recent literature in the field.

The overview can be found in Table 4.2. A distinction is made between character-level models, word-level models and models that are applied on both levels. At the bottom, three different applications are listed that have used RNNs to model various sequential problems. We immediately notice that only 5 out of the 22 investigated papers explicitly mention training details regarding loss aggregation or hidden state propagation. In the other cases we had to go through the source code manually to infer the training and sampling scheme. If there was no source code available, we contacted

Table 4.2: Concise literature overview on RNN-based language models.

Reference	Model type	Scheme	Information source
(Graves, 2013) [3]	Character-level	3	Author communication
(Wu, 2016) [21]	Character-level	Unknown	
(Ha, 2016) [22]	Character-level	Unknown	
(Cooijmans, 2016) [23]	Character-level	3	Author communication
(Krause, 2016) [24]	Character-level	4	Author communication
(Chung, 2016) [25]	Character-level	4	Paper
(Mujika, 2017) [26]	Character-level	4	Paper
(Zilly, 2017) [27]	Word- & character-level	4	Source code
(Melis, 2017) [28]	Word- & character-level	4	Paper
(Mikolov, 2012) [29]	Word-level	3	Source code
(Zaremba, 2014) [30]	Word-level	4	Paper
(Kim, 2015) [2]	Word-level	4	Source code
(Gal, 2016) [31]	Word-level	3	Source code
(Merity, 2016) [11]	Word-level	2/4 (?)	Author communication
(Bradbury, 2016) [32]	Word-level	4 (?)	Author communication
(Zoph, 2016) [33]	Word-level	4	Author communication
(Inan, 2016) [34]	Word-level	4	Source code
(Merity, 2017) [35]	Word-level	4	Source code
(Yang, 2017) [36]	Word-level	3	Author communication
(Sturm, 2016) [37]	Music notes	3	Author communication
(Saon, 2016) [38]	Phonemes	3	Author communication
(De Boom, 2017) [39]	Playlist tracks	2	Paper

the authors directly to ask for more details. Whenever we could not find information in the paper, the source code or through the authors, we have marked it with 'Unknown'.

Scheme number 4 is by far the most popular in recent literature, but scheme number 3 is also widely used. As mentioned previously, the main difference between these two schemes is whether the transfer of the hidden state between subsequent training sequences takes place or not. There seems to be no clear consensus on this topic among researchers. The older works from 2012 and 2013 by Graves [3] and Mikolov et al. [29] (and by extension, most of the older works on RNNs) do not transfer the hidden state, while the community seems to be transitioning towards explicitly doing this. Although there exists no literature describing the advantages and disadvantages of both methods, we can think of some possible explanations for this. First, while going through multiple source code repositories, we have noticed that source code is often reused by copying and adapting from previous work. This causes architectural and computational designs to transfer from previous work into other works. Another possible cause lies with the evolution of deep learning frameworks. Tensorflow, Keras and PyTorch have made it fairly easy to train RNNs with hidden state transfer, while this was less straightforward or required more effort in older frameworks, such as Theano and Lasagne.

To conclude this concise overview, we have shown that there is a need for clarity and transparency in literature concerning training and sampling details for RNNs. Not only in the interest of reproducibility, but also to spike awareness in the research community. This paper is a first attempt at calling attention to the different training and sampling schemes for RNNs, and which trade-offs each of these pose. In the next section, each of the schemes is evaluated thoroughly in a number of experimental settings.

4.4 Evaluation

In this section, all training and sampling schemes are evaluated in a variety of settings. As mentioned before, the task in each of these settings is the same: predicting the next token or character in a sequence given the previous tokens or characters. To perform the evaluation we will use four datasets with different characteristics: English text, Finnish text, C code, and classical music. Next to this, we will vary the RNN architecture—such as the number of recurrent layers and the hidden state size—as well as the truncated BPTT parameters. Through these evaluations, we will give some recommendations on how to train and sample from character-level RNNs.

4.4.1 Experimental setup

The central part of our experiments is the RNN model. For this, we construct a standard architecture with some parameters that we can vary. The input of the RNN is a one-hot representation of the current character in the sequence, i.e. a vector of zeros with length $|V|$, with V the ordered set of all characters in the dataset, except for a single one at the current character's position in V. Next, r recurrent LSTM layers are added, each with a hidden state dimensionality of γ. In the experiments, the parameters of r and γ will be varied. Finally, we add two fully connected dense layers, one with a fixed dimensionality of 1,024 neurons, and the final dense layer again has dimensionality $|V|$. At this final layer a softmax function is applied in order to arrive at a probability distribution across all possible next characters. The complete architecture is summarized in Table 4.3 including nonlinear activation functions and extra details.

To train the RNN model we will use one of the schemes outlined in Section 4.3. As is common practice in deep learning and gradient-based optimization, multiple training sequences are grouped in batches. Each sequence in such a batch has a length of $k_2 + 1$ tokens, from which the first k_2 are used as input to the RNN, and the next token is used as ground truth signal for every input token. In this paper, a batch size of 64 sequences is used across all experiments. To ensure a diverse mix of sequences in each batch, we pick sequences at equidistant offsets, which we increase by k_1 for every new batch. More specifically, every i'th batch 64 sequences are sampled at the following offsets in the train set $\mathcal{D}_{\text{train}}$:

$$\frac{j \cdot \text{size}(\mathcal{D}_{\text{train}})}{64} + i \cdot k_1,\ j \in \{0, 1, \ldots, 63\}. \tag{4.8}$$

The entire train set is also circularly shifted after each training epoch. Since in scheme 4 the hidden states is transferred across different batches, this batching method allows us to fairly compare all four schemes.

At regular points during training the performance of the RNN is evaluated with data from the test set $\mathcal{D}_{\text{test}}$. For this we will use the perplexity measure, which is widely used in evaluating language models:

$$\text{perplexity} = \exp\left(\frac{-\sum_{i=1}^{N} \log p(x_i | x_{1:i-1})}{N}\right). \tag{4.9}$$

In this formula x_i is the i'th token in the sequence and N is the total number of tokens. The better a model is at predicting the next token, the lower its perplexity measure. In the context of RNNs, the conditional probability in Equation (4.9) is approximated using the hidden state of the RNN, as was

Table 4.3: The RNN architecture used in all experiments.

	Layer type (no. of dimensions) and nonlinearity
	Input ($\|V\|$)
1 to r	LSTM (γ)
	sigmoid (gates); tanh (hidden and cell state update)
	orthogonal initialization, gradient clipping at 50.0
$r+1$	Fully connected dense (1,024)
	leaky ReLU, leakiness $= 0.01$, glorot uniform initialization
$r+2$	Fully connected dense ($\|V\|$)
	softmax, glorot uniform initialization

shown in Equation (4.3). In practice, the hidden state of the RNN is bootstrapped with k_2 characters and perplexity is calculated on all subsequent characters in the test set.

In the experiments below, every RNN is trained with 12,800 batches of 64 sequences using the batching method described above. For all schemes and experiments the standard categorical cross-entropy loss function is used, which calculates the inner product between the log output probability vector $\mathbf{y}$ and the one-hot vector of the target token $\hat{x}$:

$$\mathcal{L}(\mathbf{y}, \hat{x}) = -\mathbf{onehot}(\hat{x}) \cdot \log(\mathbf{y}) . \tag{4.10}$$

During training we report perplexity on the test set at logarithmically spaced intervals. All RNN models are trained five times with k_2 always set to 100 (unless explicitly indicated otherwise), and we choose $k_1 \in \{20, 40, 60, 80, 100\}$. For every new configuration we reinitialize all network weights and random generators to the same initial values. As optimization algorithm we use Adam with a learning rate of 0.001 throughout the experiments.

All experiments are performed on a single machine, 12 core Intel Xeon E5-2620 2.40GHz CPU and Nvidia Tesla K40c GPU. We use a combination of Theano 0.9 and Lasagne 0.2 as implementation tools, powered by cuDNN 5.0.

4.4.2 Datasets

In the experiments the performance of each scheme is evaluated on four datasets[1]. We list the dataset characteristics below:

[1]The datasets are available for download at https://github.com/cedricdeboom/character-level-rnn-datasets

1. English: we compiled all plays by William Shakespeare from the Project Gutenberg website[2] in one dataset. The plays follow each other in random order. The total number of characters is 6,347,705 with 85 unique characters.

2. Finnish: this language is very different from English. On the Gutenberg website we gathered all texts from Finnish playwrights Juhani Aho and Eino Leino. This results in a dataset of 10,976,530 characters, of which 106 are unique.

3. Linux: we saved all C code from the Linux kernel[3] and gathered the files together. On November 22 2016, the entire kernel contained 6,546,665 characters, and 97 of them are unique.

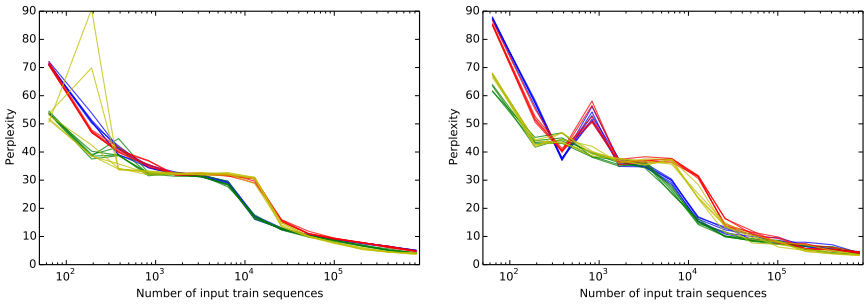
4. Music: we created this dataset by extracting music notes from MIDI files. When notes are played simultaneously in the MIDI file, we extract them from low to high, so that we obtain a single sequence of subsequent notes. We downloaded all piano compositions by Bach, Beethoven, Chopin and Haydn from Classical Archives[4], removed duplicate compositions, and gathered a dataset of 1,553,852 notes, of which there are 90 unique ones.

After cyclically permuting each dataset over a randomly chosen offset, we extract the last 11,100 characters to compile a test set $\mathcal{D}_{\text{test}}$. All remaining characters form the train set $\mathcal{D}_{\text{train}}$.

4.4.3 Experiments

Several experiments are now performed to evaluate the predictive performance of RNNs that have been trained with different configurations. In a first round of experiments the architecture of the RNN models is varied. More specifically, we set the number of recurrent LSTM layers r to 1 or 2, and we also change the hidden state size γ to either 128 or 512. Figure 4.7 shows plots for these different RNN architectures, trained using scheme 1 and on all four datasets. For every architecture we have plotted five lines for the different settings of k_1 mentioned above. In all plots we see that the RNNs with $\gamma = 512$ (green and yellow) initially perform better, but the RNNs with $r = 1$ (green and blue) learn somewhat faster in the long term. At 12,800 batches there is no clear difference in performance anymore between the architectures. On the music dataset and architectures

[2] www.gutenberg.org
[3] github.com/torvalds/linux/tree/master/kernel
[4] www.classicalarchives.com



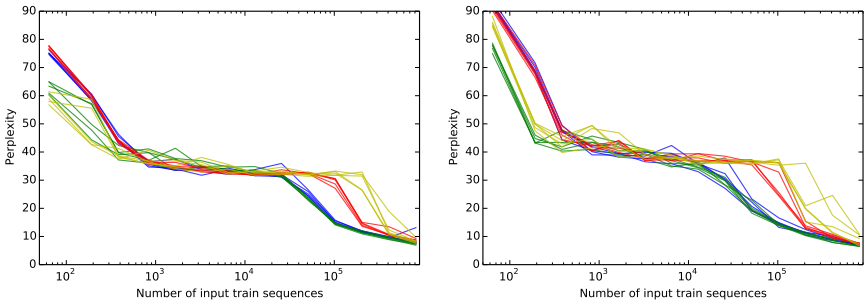
(a) English (b) Finnish

(c) Linux (d) Music

Figure 4.7: Comparing RNN architectures with scheme 1. Blue: $r = 1, \gamma = 128$; Green: $r = 1, \gamma = 512$; Red: $r = 2, \gamma = 128$; Yellow: $r = 2, \gamma = 512$.

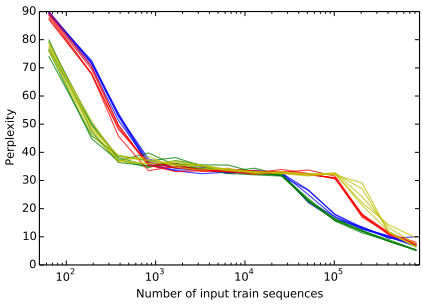
with $\gamma = 512$ we observe some overfitting. If we add 25% of dropout to the final two dense layers [43], this overfitting is already greatly reduced, but still observable (not shown in the graph). For the Finnish dataset and architectures with $\gamma = 128$ we notice a bump around 1,000 train sequences, which is present for all k_1 configurations. This bump is lowered if we reduce the learning rate to 0.0001 or use a different, non momentum-based optimizer such as RMSProp, but it remains an artefact of both the dataset and architecture.

We also perform the same experiments with scheme 2, for which the results are shown in Figure 4.8. The same behavior with respect to the architectural differences is observed as in the first scheme. But now the networks converge somewhat slower, which can be seen especially for the Music dataset by comparing Figures 4.7d and 4.8d. On the plus side, the performance curves are smoother than for scheme 1. Both effects can be explained by the fact that there is only one loss signal at the end of each training sequence, which makes learning slower, but the backpropagated

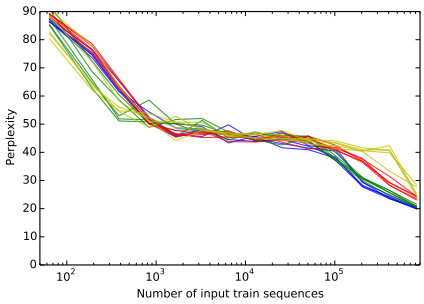


(a) English

(b) Finnish



(c) Linux

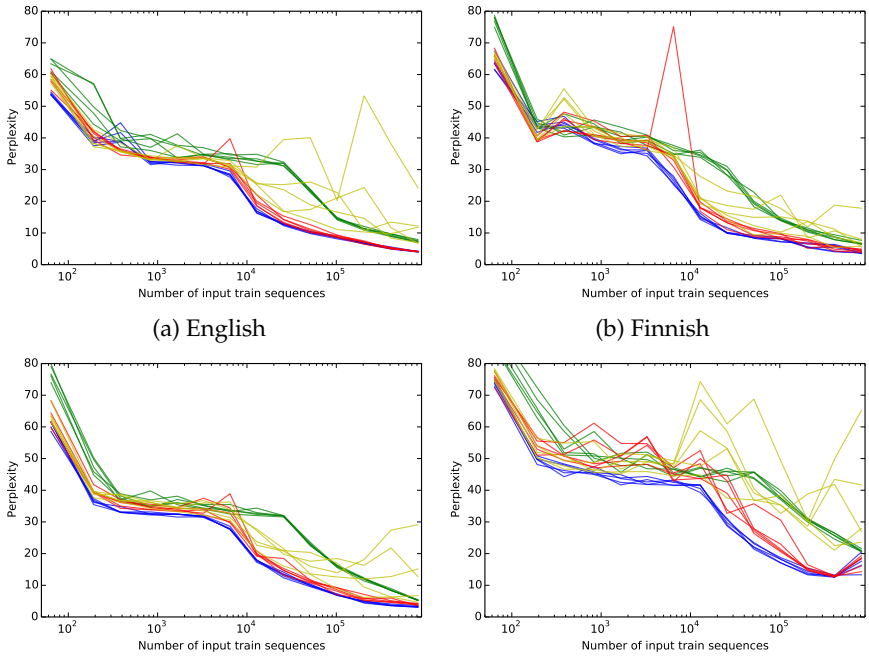


(d) Music

Figure 4.8: Comparing RNN architectures with scheme 2. Blue: $r = 1, \gamma = 128$; Green: $r = 1, \gamma = 512$; Red: $r = 2, \gamma = 128$; Yellow: $r = 2, \gamma = 512$.

gradient is of higher quality, since more historical characters are taken into account. From Figures 4.7 and 4.8 we conclude that the RNN architecture indeed influences the efficiency of the training procedure, but that the same effect is observed globally across datasets and training schemes. The best architecture for all four datasets has parameters $r = 1$ and $\gamma = 512$, i.e. the green plots. This specific architecture will therefore be used in the next experiments.

Next, all schemes are compared on the different datasets. As mentioned above, the architecture with $r = 1, \gamma = 512$ is used. The perplexity plots are gathered in Figure 4.9. We see that schemes 1 and 2 are very robust across datasets, but also across different settings of k_1, since all lines lie very close to each other. Scheme 1 is also the best performing in terms of perplexity. The performance of scheme 2 is overall worse compared to scheme 1, which is probably due to the fact that learning occurs more slowly, as argued before. The performance of scheme 3 is comparable to the first scheme, but only very slightly worse and robust. Since the training pro-



(a) English (b) Finnish

(c) Linux (d) Music

Figure 4.9: Comparing RNN schemes. Blue: scheme 1; Green: scheme 2; Red: scheme 3; Yellow: scheme 4.

cedure of schemes 1 and 3 is the same, we hypothesize that the sampling procedure of scheme 3 sometimes has difficulties recovering from errors, which can be carried across many time steps. Another reason is that the RNN has not learned to make predictions for sequences longer than k_2 tokens. We also mention that for schemes 1, 2 and 3 we experimented with randomly shuffling all training sequences instead of circularly shifting the train set, as explained in Section 4.4.1, but this did not lead to different observations. In scheme 4 the hidden state is transferred across sequences during training, which appears to solve this problem, at least for some configurations of k_1. All configurations for scheme 4 start with the same performance as for scheme 3, but after around 200 train batches—i.e. 12,800 train sequences in the graph—some configurations start diverging, for which we cannot isolate any consistent motivation or explanation. From the figures we see that this behavior is also heavily dependent on the dataset; the difference between e.g. the Finnish and Music dataset is notable.

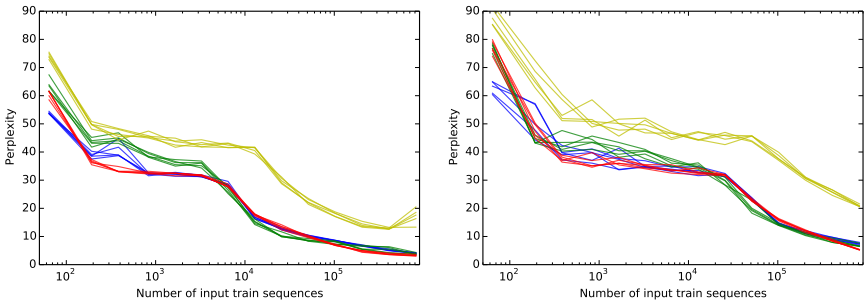
Table 4.4: Absolute training time per batch and sampling time for 1 token using different RNN configurations on the English dataset, $k_1 = 40$. Measurements in ms. The variance is negligible.

	$r = 1, \gamma = 128$	$r = 1, \gamma = 512$	$r = 2, \gamma = 128$	$r = 2, \gamma = 512$
Scheme 1	60.5 / 7.0	138.8 / 21.7	106.5 / 13.7	267.4 / 43.3
Scheme 2	46.5 / 7.0	114.6 / 21.7	93.1 / 13.7	245.8 / 43.3
Scheme 3	60.5 / 0.7	138.8 / 1.2	106.5 / 1.1	267.5 / 2.2
Scheme 4	60.6 / 0.7	138.9 / 1.2	106.6 / 1.1	267.6 / 2.2

It is also interesting to take a look at a comparison between performances on different datasets for the same scheme. These curves are plotted in Figure 4.10. For all schemes we notice that the performance on the English, Finnish and Linux datasets is almost equal; only the Music dataset seems harder to model with the same RNN architecture. What we also observe is that scheme 1 is very robust against changes in training parameters, since all curves lie very close to each other. There is more variance in this for scheme 2, even more for scheme 3, and it is highest for scheme 4.

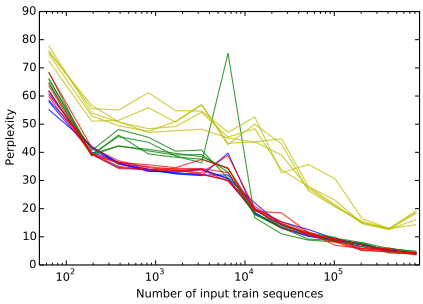
At this point we would also like to discuss data efficiency. For small values of k_1, we use less data at a particular point in the training process compared to larger values of k_1. This is important when data resources are scarce. From Figure 4.10 it is noticeable that, at least for the same scheme, the lines for different values of k_1 lie very close to each other. From these experiments, a general conclusion could be to use a small value of k_1 in order to be as data efficient as possible. The choice of k_1, after all, seems to have less impact than the choice of training scheme. Additionally, using a small value of k_1 improves label reuse in the multi-loss training algorithms. This can approximately be quantified by k_2/k_1, i.e. the number of times a label is reused in the training process.

Up until now we have been comparing the performance of different RNN models and schemes in terms of the number of train sequences used up until a certain point in time. But the models can also be compared in terms of absolute training and sampling time, which will give us an overview of which configurations are the fastest. In the next experiment, we calculate the average training time per batch and sampling time for a single token on the English dataset. We will vary the scheme that we use for training, as well as the RNN architecture. Concerning the k_1 training parameter, there will be almost no difference in training time, so in all measurements we use $k_1 = 40$. The numbers are shown in Table 4.4. It is no surprise that schemes 1, 3 and 4 have almost equal training time per batch, while scheme 2 trains significantly faster since we only need to compute one softmax output for each training sequence. It is however noticeable

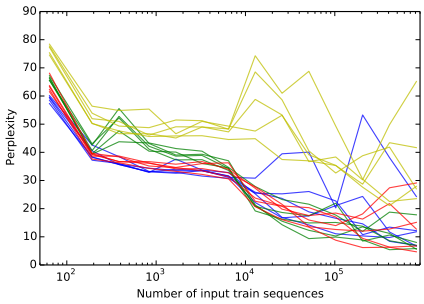


(a) Scheme 1

(b) Scheme 2



(c) Scheme 3

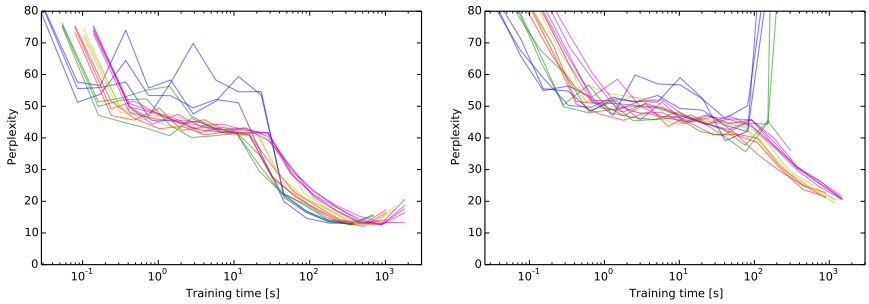


(d) Scheme 4

Figure 4.10: Comparing datasets. Blue: English; Green: Finnish; Red: Linux; Yellow: Music.

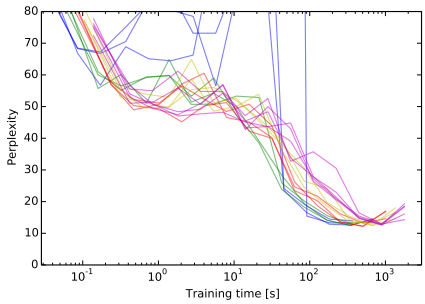
that the more complex the RNN architecture, the smaller the relative difference in training time, with a decrease of 25% for the $r = 1, \gamma = 128$ architecture and just 9% for the $r = 2, \gamma = 512$ architecture. Regarding the sampling times, we see that the 3rd and 4th schemes are faster by a factor of 10 up to 20 compared to schemes 1 and 2, since there is no need to propagate an entire sequence through the RNN to sample a new token.

We also compare the performance of the different schemes with respect to changes in the k_2 parameter. For each scheme we perform five experiments, for which k_2 is set successively to 20, 40, 60, 80 and 100. After setting k_2, the k_1 parameter is set to k_2, $2k_2/3$ and $k_2/3$, rounded to the nearest integer. Every experiment is performed on the Music dataset, since, based on previous experiments, we expect to gain most insights on it. We report the model perplexity on the test set as a function of the elapsed training time, and we train again for a total of 12,800 batches. The results are shown in Figure 4.11, in which the y-axis is clipped to a maximum of 80 to achieve the most informative view. We see that the smaller the k_2 value, the

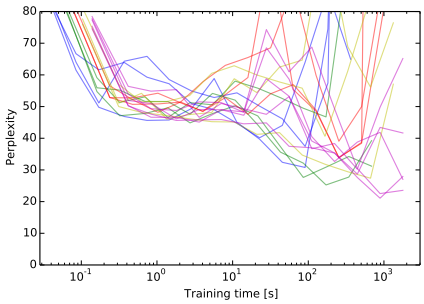


(a) Scheme 1

(b) Scheme 2



(c) Scheme 3



(d) Scheme 4

Figure 4.11: Performance with respect to training time by comparing k_2 values on the Music dataset. Blue: $k_2 = 20$; Green: $k_2 = 40$; Red: $k_2 = 60$; Yellow: $k_2 = 80$; Magenta: $k_2 = 100$.

faster we have trained all batches, since it leads to a shorter BPTT. The first scheme is again the most robust against a changes in k_2. Only the shortest sequence lengths behave more noisily in the first 10 seconds of training, but all configurations are able to reach a similar optimal perplexity. The second scheme trains much slower than scheme 1, and experiences instability problems for small sequence lengths of 20 and 40. The configurations with $k_2 \geq 60$ are all very stable, but have not yet fully converged after 12,800 batches. For scheme 3 we see almost the same behavior as in scheme 1, with all configurations reaching the same optimal perplexity. But, just as we saw before, the robustness against changes in k_1 is worse. This is especially true for small values of k_2, as shown by the blue lines in Figure 4.11c. Finally, for scheme 4 we see that almost all configurations are unstable and behave very noisily. Two configurations with $k_2 = 20$ even achieve a final perplexity of around 350; lowering k_1 for small values of k_2 seems to help in this case.

4.4.4 Take-away messages

We conclude this experimentation section with a few recommendations. We found that the global behavior of the different schemes is nearly independent of the used dataset. This is good news, since we do not have to tune the learning and sampling procedure to the dataset at hand. In this respect, we arrive at the following conclusions:

- In terms of training schemes, the multi-loss approach (scheme 1 and 3) is recommended. Compared to the single-loss approach (scheme 2), multi-loss training is more efficient. The faster individual iterations of the single-loss approach cannot compensate for the benefit of combining the loss over multiple positions in the sequence, when considering the total train time.
- Our general recommendation is to avoid training procedures in which the hidden state is transferred between input sequences (scheme 4). Training is as efficient as the multi-loss approach without transferred hidden states (scheme 3), but less robust. On noisy datasets, such as the Music dataset in our experiments, transferring hidden states is likely to cause an unstable behavior.
- On the sampling side, there is a trade-off between windowed sampling and progressive sampling. By comparing scheme 1 and 3, it is seen that windowed sampling is more robust than progressive sampling. However, the latter is more efficient by construction, as it samples the next character based on the current one and the hidden state, instead of each time performing a forward pass over a (possibly long) sequence as in the windowed sampling approach.

4.5 Future research tracks

We include one final section on future research tracks in the area of training and sampling procedures for character-level RNNs. In this paper we have made an attempt at isolating the four most common schemes that have been or are being used in literature. There are however multiple hybrid combinations that can be identified and investigated in the future. The most straightforward extension is an intermediate form between single- and multi-loss training. For example, an extra parameter k_3 could be identified, for which $k_3 \leq k_2$, that defines the number of time steps for which the loss is calculated and aggregated. The edge cases $k_3 = 1$ and $k_3 = k_2$ correspond respectively to the single-loss and multi-loss training procedures. One other possibility is to decay the loss at each time step (linearly

or exponentially) and combine these through a linear combination to calculate the final loss. For a single training sequence s this results in:

$$\texttt{loss}_s = \sum_{(x_i, x_{i+1}) \in s} \mathcal{L}\left(\mathcal{R}(x_i), x_{i+1}\right) \cdot \gamma_i,$$

with $\gamma_i = {}^{i}/_{\text{length}(s)-1}$ or $\gamma_i = \exp(i - \text{length}(s) + 1)$ for resp. linear and exponential decay. Consequentially, the resulting gradient is scaled similarly, thereby reducing the contribution of the first few tokens in the sequence to the total loss.

4.6 Conclusion

We explained the concept of character-level RNNs and how such models are typically trained using truncated backpropagation through time. We then introduced four schemes to train character-level RNNs and how to sample new tokens from such models. These schemes differ in how they approximate the truncated backpropagation through time paradigm: how the RNN outputs are combined in the final loss, and whether the hidden state of the RNN is remembered or reset for each new input sequence. After that, we evaluated each scheme against different datasets and RNN configurations in terms of predictive performance and training time. We showed that our conclusions remain valid across all these different experimental settings.

Perhaps the most surprising result of the study is that conditional multi-loss training, in which the hidden state is carried across training sequences, often leads to unstable training behavior depending on the dataset. This contrasts sharply with the observation that this training procedure is used most often in literature, although it requires meticulous bookkeeping of the hidden state and a carefully designed batching method. Single-loss training is, compared to multi-loss, slower regarding the number of used train sequences. An advantage of single-loss training, however, is that we encourage the network to make predictions on a long-term basis, since we only backpropagate one loss defined at the end of a sequence.

We saw that progressive sampling is slightly less robust to changes in training parameters compared to windowed sampling, especially for datasets that are more difficult to model, as we showed with the Music dataset. The main advantage of progressive sampling is that it is orders of magnitudes faster than windowed sampling.

References

[1] K. Gregor, I. Danihelka, A. Mnih, C. Blundell, and D. Wierstra. *Deep AutoRegressive Networks*. arXiv.org, October 2013. arXiv:1310.8499v2.

[2] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush. *Character-Aware Neural Language Models*. arXiv.org, August 2015. arXiv:1508.06615v3.

[3] A. Graves. *Generating Sequences With Recurrent Neural Networks*. arXiv.org, August 2013. arXiv:1308.0850v5.

[4] A. Karpathy, J. Johnson, and L. Fei-Fei. *Visualizing and Understanding Recurrent Networks*. arXiv.org, June 2015. arXiv:1506.02078v2.

[5] T. Sercu and V. Goel. *Advances in Very Deep Convolutional Neural Networks for LVCSR.* In Interspeech, 2016.

[6] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. *WaveNet: A Generative Model for Raw Audio*. arXiv.org, September 2016. arXiv:1609.03499v2.

[7] K. Gregor, I. Danihelka, A. Graves, and D. Wierstra. *DRAW: A Recurrent Neural Network For Image Generation*. arXiv.org, February 2015. arXiv:1502.04623v1.

[8] A. v. d. Oord, N. Kalchbrenner, and K. Kavukcuoglu. *Pixel Recurrent Neural Networks*. January 2016. arXiv:1601.06759.

[9] Y. K. Tan, X. Xu, and Y. Liu. *Improved Recurrent Neural Networks for Session-based Recommendations.* arXiv.org, 2016. arXiv:1606.08117v2.

[10] B. Hidasi, A. Karatzoglou, L. Baltrunas, and D. Tikk. *Session-based Recommendations with Recurrent Neural Networks.* arXiv.org, 2016. arXiv:1511.06939v4.

[11] S. Merity, C. Xiong, J. Bradbury, and R. Socher. *Pointer Sentinel Mixture Models*. arXiv.org, September 2016. arXiv:1609.07843v1.

[12] I. Sutskever. *Training recurrent neural networks*. PhD thesis, 2013.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning representations by back-propagating errors*. Cognitive modeling, 1988.

[14] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[15] D. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. In ICLR, 2015.

[16] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*. 2001.

[17] S. Hochreiter and J. Schmidhuber. *Long short-term memory*. Neural Computation, 9(8):1735–1780, 1997.

[18] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. *LSTM: A Search Space Odyssey*. arXiv.org, March 2015. arXiv:1503.04069v1.

[19] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. arXiv.org, 2014. arXiv:1412.3555v1.

[20] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. arXiv.org, June 2014. arXiv:1406.1078v3.

[21] Y. Wu, S. Zhang, Y. Zhang, Y. Bengio, and R. Salakhutdinov. *On Multiplicative Integration with Recurrent Neural Networks*. arXiv.org, June 2016. arXiv:1606.06630v2.

[22] D. Ha, A. Dai, and Q. V. Le. *HyperNetworks*. arXiv.org, September 2016. arXiv:1609.09106v4.

[23] T. Cooijmans, N. Ballas, C. Laurent, and A. Courville. *Recurrent Batch Normalization*. arXiv.org, March 2016. arXiv:1603.09025v5.

[24] B. Krause, L. Lu, I. Murray, and S. Renals. *Multiplicative LSTM for sequence modelling*. arXiv.org, September 2016. arXiv:1609.07959v3.

[25] J. Chung, S. Ahn, and Y. Bengio. *Hierarchical Multiscale Recurrent Neural Networks*. arXiv.org, September 2016. arXiv:1609.01704v7.

[26] A. Mujika, F. Meier, and A. Steger. *Fast-Slow Recurrent Neural Networks*. arXiv.org, May 2017. arXiv:1705.08639v2.

[27] J. G. Zilly, R. K. Srivastava, J. Koutník, and J. Schmidhuber. *Recurrent Highway Networks.* ICML, 2017.

[28] G. Melis, C. Dyer, and P. Blunsom. *On the State of the Art of Evaluation in Neural Language Models.* CoRR, 2017.

[29] T. Mikolov and G. Zweig. *Context dependent recurrent neural network language model*. In 2012 IEEE Spoken Language Technology Workshop (SLT, pages 234–239. IEEE, 2012.

[30] W. Zaremba, I. Sutskever, and O. Vinyals. *Recurrent Neural Network Regularization*. arXiv.org, September 2014. arXiv:1409.2329v5.

[31] Y. Gal and Z. Ghahramani. *A Theoretically Grounded Application of Dropout in Recurrent Neural Networks.* NIPS, 2016.

[32] J. Bradbury, S. Merity, C. Xiong, and R. Socher. *Quasi-Recurrent Neural Networks*. arXiv.org, November 2016. arXiv:1611.01576v2.

[33] B. Zoph and Q. V. Le. *Neural Architecture Search with Reinforcement Learning*. CoRR, 2016.

[34] H. Inan, K. Khosravi, and R. Socher. *Tying Word Vectors and Word Classifiers - A Loss Framework for Language Modeling*. CoRR, cs.LG, 2016.

[35] S. Merity, N. S. Keskar, and R. Socher. *Regularizing and Optimizing LSTM Language Models.* CoRR, 2017.

[36] Z. Yang, Z. Dai, R. Salakhutdinov, and W. W. Cohen. *Breaking the Softmax Bottleneck: A High-Rank RNN Language Model*. arXiv.org, November 2017. arXiv:1711.03953v2.

[37] B. L. Sturm, J. F. Santos, O. Ben-Tal, and I. Korshunova. *Music transcription modelling and composition using deep learning*. arXiv.org, April 2016. arXiv:1604.08723v1.

[38] G. Saon, T. Sercu, S. Rennie, and H.-K. J. Kuo. *The IBM 2016 English Conversational Telephone Speech Recognition System*. arXiv.org, April 2016. arXiv:1604.08242v2.

[39] C. De Boom, R. Agrawal, S. Hansen, E. Kumar, R. Yon, C.-W. Chen, T. Demeester, and B. Dhoedt. *Large-scale user modeling with recurrent neural networks for music discovery on multiple time scales*. Multimedia Tools and Applications, pages 1–23, August 2017.

[40] M. Marcus, B. Santorini, and M. A. Marcinkiewicz. *Building a large annotated corpus of English: the Penn Treebank*. Computational Linguistics, 19(2), 1993.

[41] T. Mikolov, M. Karafiát, L. Burget, J. Cernocky, and S. Khudanpur. *Recurrent neural network based language model*. In Interspeech, 2010.

[42] M. Hutter. *The Human Knowledge Compression Contest* [online]. 2012. Available from: http:/prize.hutter1.net.

[43] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. *Dropout - a simple way to prevent neural networks from overfitting.* Journal of Machine Learning Research, 15:1929–1958, 2014.

5

Polyphonic Piano Music Composition with Composer Style Injection using Recurrent Neural Networks

"Please, don't stop the music."

—Rihanna, 2007

Now that we have a more than basic understanding of the training and sampling concepts behind recurrent neural networks, we will spend the next two chapters applying them in various applications. In this chapter, we will explore the research field of automatic music composition. Based on the idea of the character-level RNN from Chapter 4, we will compose novel music one note after the other. That is, the model that we will train is able to make a prediction for the next note in a composition based on the notes that have already been written down. We will also explore whether it is possible to tune the composition process by considering a composer's identity: will a model trained on Beethoven music sound different than a model that is trained on piano pieces by Chopin? And is it possible to change this composer information on the fly?

⋆ ⋆ ⋆

C. De Boom, M. De Coster, D. Spitael, S. Leroux, S. Bohez, T. Demeester, and B. Dhoedt

Submitted to Neural Computing and Applications, December, 2017.

Abstract Automated music composition can teach us a lot about how humans perceive certain musical properties, but it also has useful applications in entertainment and cinematic industries. In these areas the musical score often needs to be adjusted to a specific context, mood or scene. There, instead of manually composing or selecting the appropriate music, we would like to be able to steer the process of automatic music generation using style or mood information. For this purpose we constructed a deep learning model that allows for musical style injection during composition. In particular, this paper describes how we use recurrent neural networks to model polyphonic piano music, and how we adjust the network architecture to incorporate style information from four stylistically distinct composers: Bach, Haydn, Beethoven and Chopin. We have conducted field studies in a target audience with a background in classical music, and we found that our models are capable of generating plausible polyphonic compositions, and that the style of the music can be altered on-the-fly whilst tuning the composer identity during composition. The generated musical pieces are, however, still discernible from hand-made compositions.

5.1 Introduction

The quest for computer-aided music composition has always been regarded as both difficult and intriguing among researchers. The idea that we can teach the art of creativity to a computer and to have a never-ending and always changing stream of the best and most pleasing music at one's disposal, is very tantalizing. Although most of the musical pieces we listen to on a daily basis obey a rather compact set of harmonic and rhythmical guidelines and rules, the main challenge often lies in capturing an overall coherent musical style, logical phrasings, appealing melodies and themes, a notion of surprise, and of course a global structure for the song. A single model or algorithm that successfully tackles all these challenges at once, does not yet exist, but recently major efforts regarding this subject have been made by various research institutions.

Ever since the 1980s researchers have tackled automatic music generation. On the one hand we have symbolic music generation, in which we compose a piece of music as if written out on sheet music paper, and acous-

tic music generation on the other hand, where we directly generate digital audio signals. This paper focuses on symbolic music composition, since generating raw audio has proven to be a computationally and resource demanding effort and is still considered a very hard task to date [1, 2]. The first musical composition algorithms can nowadays be classified as rule- and template-based systems. They were often highly engineered using various heuristics, but this made them quite easily interpretable for humans. For example, in the Experiments in Musical Intelligence (EMI) by Cope a collection of templates are used to transfer the melodies and harmonies from one piece of music onto the rhythmic structure of another—although rigorous details on the methodology are absent in literature [3]. In 1990, the CHORAL system by Ebcioğlu defined a set of highly detailed rules to harmonize Bach chorales [4]. More recently, in 2015, Brown et al. engineered a collection of heuristics that rest on the psychological grouping, or 'Gestalt', principles of proximity, similarity, closure, continuity and connectedness [5]. These heuristics are then used to generate melodies in real time. There have also been attempts at composing polyphonic counterpoint music using carefully crafted metaheuristics and objective functions, which are optimized through a neighborhood search algorithm [6].

Despite the results that were obtained with the above systems, a lot of manual work and insights about musical and stylistic properties are required to get adequate and pleasing results. These days, artificial intelligence research is mainly about learning an appropriate model directly from the data itself with little or no human intervention. If these models are of 'sufficiently high complexity', and given enough training data, the underlying structure and patterns that are present in the data can automatically be captured, up to a certain degree. In the context of symbolic music generation, a popular method that is most often used in literature, is a note per note prediction scheme. The first attempts at this primarily made use of stochastic processes based on transition tables, such as Markov models [7, 8]. For an n^{th} order Markov model, the next note in the piece is probabilistically determined based on the sequence of the previous n notes, i.e. the historical context. The transition probabilities in these Markov models are learned by processing a large data collection. Through the use of n-grams in Markov models, Ponsford et al. [9] were able to learn context-free grammars for harmonic movement in 17th-century sarabandes. Schulze et al. used multiple hidden Markov models to decouple chord progressions, durations and melodic arcs [10]. In 2014, Pachet et al. from the group behind Sony's Flow Machines also used Markov models to arrange jazz standards reminiscent of a specific harmonization style [11]. As

all these works show, Markov models can be effective at modeling certain musical aspects. But since they are inherently focused on local transitions, it is still difficult to grasp the global structure and overall coherence of a song without manual engineering.

Neural networks, on the other hand, have a greater and more flexible representational power than Markov models. Recurrent neural networks (RNNs), a particular subset of neural networks, are specifically designed to process symbolic sequences, are not limited to a fixed context window, and are able to model local as well as global structure in a sequence. In 1989, Todd was among the first to generate monophonic melodies using basic RNNs [12], and in 1994, Mozer generated melodies with chord accompaniments using RNNs and psychoacoustically motivated pitch and chord representations [13]. The author stated that the generated music was very discernible from real compositions and lacked structure, but this work was done well before the advent of more powerful RNN architectures, such as LSTMs [14] and GRUs [15], 'deep' networks and highly scalable hardware infrastructure. Eck and Schmidhuber were the first to use LSTMs to generate blues improvisations with chord progressions and reported that their model was able to successfully learn the global structure of a song [16].

Almost all previous work in deep learning for music generation mainly focused on monophonic music in which only one note sounds at a time. Most music, however, is polyphonic where multiple notes and sounds are played simultaneously to arrive at rich harmonies. The main difficulty in polyphonic music generation is that it requires multiple decisions at the same time that can all influence each other, thereby rendering it a so-called 'structured prediction' problem. For example, predicting the different pitches in a single chord independently from each other might lead to ill-sounding music. Boulanger-Lewandowski et al. focused on this particular problem for which they used a combination of RNNs and restricted Boltzmann machines, a type of energy-based neural prediction models [17]. Since very recently, music generation with deep RNNs, and especially LSTM-based architectures, has gained great interest again [18, 19]. Jaques et al. used reinforcement learning with rewards based on music theory to finetune a note-based RNN [20]. Choi et al. generated chord progressions and drum tracks with RNNs [21]. The latter use-case, however, failed since the authors did not foresee that it requires a structured prediction scheme. In the work by Walder this issue is tackled by solving the structured decision problem in a sequential manner: notes that sound simultaneously are predicted one after the other, and the next note is conditioned on all previous ones [22]. A similar idea is adopted in this paper, for which we will

use RNNs to summarize the past note sequence and use this summary to make a conditioned prediction about the next note in the piece.

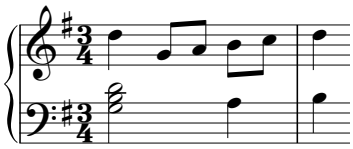
Finally, we also mention that regarding acoustic music generation, WaveNet by van den Oord et al. was the first to produce ground-breaking and realistically sounding results [1]. The authors also introduced the concept of a conditional WaveNet, with which they were able to learn a single speech model for multiple speakers by conditioning the generation process on the speaker identity. In this paper, we will use a similar approach to generate symbolic music in the style of a given composer through one single model.

In spite of all the recent advances, generating realistic and naturally sounding music remains difficult and challenging. This paper will not provide an answer to all the research questions posed at the beginning of this introduction. Instead we will focus on one aspect of the music composition process, which is how we can capture the identity of a composer and use that identity to drive the music generation. In this paper we will focus on differentiating between four composers that each lived in their own distinct cultural period: Bach, Haydn, Beethoven and Chopin. A human music expert should already be able to identify the correct composer with a fairly high precision, even from a small excerpt of music: Bach's music is representative for the baroque period, Haydn is a classical composer from the same period as Mozart, Beethoven marked the beginning of the romantic period in which also Chopin lived. We therefore want to verify whether a music generator would also be able to produce music with certain composer-specific properties. And once we have such a model, we want to know if it is possible to generate music in which we gradually switch from one style to another. Although we do not show this explicitly in this paper, our technique is directly applicable to musical styles other than composer identity, e.g. the genre (waltz, tango, bossa nova…), mood (peaceful, dramatic…), instrumentation, etc. With these techniques we could then for example generate a soundtrack for a movie and continuously tune the music to the mood of each scene on-the-fly, similar to what Malleson et al. did for movie actor emotions [23].

5.2 Problem setting

Given a piece of music consisting of N notes, each note a_i in this piece is fully characterized by three numerical quantities: its pitch p_i, duration d_i and start time s_i:

$$\forall i \in \{1, 2, \dots N\} : a_i = (p_i, d_i, s_i). \tag{5.1}$$



i	p_i	d_i	s_i
1	G_3	1/2	0
2	B_3	1/2	0
3	D_4	1/2	0
4	D_5	1/4	0
5	G_4	1/8	2/8
6	A_4	1/8	3/8
7	A_3	1/4	4/8
8	B_4	1/8	4/8
9	C_5	1/8	5/8
10	B_3	1/4	6/8
11	D_5	1/4	6/8

Figure 5.1: Example representation according to Equations (5.1)–(5.3) for the short piece of music on the left. The quantities d_i and s_i are displayed in symbolic note length.

We regard an entire musical composition as a sequence of N notes that are ordered according to their start times:

$$\forall(a_i, a_{i+1}) \subset a_{1:N} \colon s_i \leq s_{i+1}. \tag{5.2}$$

In this we used the colon notation $a_{j:k}$ to indicate the range $(a_j, a_{j+1}, \ldots, a_k)$. This representation is in line with the Prolog format in [9]. A chord is a group of notes that are played simultaneously. It is formally defined as a subsequence of notes that have the same start time and are sorted according to pitch:

$$\begin{aligned} a_{j:k} \text{ is a chord } \Leftrightarrow \forall i \in \{j, j+1, \ldots k-1\} \colon \\ s_i = s_{i+1} \wedge p_i < p_{i+1}. \end{aligned} \tag{5.3}$$

If the series of notes $a_{j:k}$ in $a_{1:N}$ all have the same start time but do not form a valid chord according to Equation (5.3), we rearrange the notes in $a_{j:k}$ until it is a valid chord. Figure 5.1 shows how a short piece of music can be transformed into the representation that we have just sketched.

The problem of symbolic music generation is now modeled as a note per note prediction task. That is, the next note a_i in a musical piece is sampled conditioned on the previous history $a_{1:i-1}$:

$$a_i \sim P(a_i | a_{1:i-1}). \tag{5.4}$$

The entire generated musical composition can then be factorized as follows:

$$P(a_{1:N}) = P(a_1) \prod_{i=2}^{N} P(a_i | a_{1:i-1}). \tag{5.5}$$

Note that, in this framework, the structured problem of predicting notes that are played simultaneously is modeled in a sequential manner, by analogy with Walder [22]. A note in a chord, for example, is therefore always conditioned on all simultaneous notes with a lower pitch.

The music generation process can also be conditioned on the desired style. That is, the next note is predicted not only based on the previous history, but we also give additional information on how we want the music to sound. This information can be kept constant to arrive at a single coherent piece of music, but it is also possible to vary the information across the piece. For each note a_i we capture these additional style or context parameters in the quantity $\mathbf{c}_i$, after which we arrive at the following factorization:

$$P(a_{1:N} | \mathbf{c}_{1:N}) = P(a_1 | \mathbf{c}_1) \prod_{i=2}^{N} P(a_i | a_{1:i-1}, \mathbf{c}_i). \tag{5.6}$$

5.3 Methodology

In this section we will outline the details regarding the entire music generation process. First, we have to define a suitable and useful representation for sequences of musical notes. After that, a model has to be specified in order to learn to generate such sequences in terms of the probability distributions in Equations (5.5) and (5.6).

5.3.1 Representing musical notes

In Equation (5.1) we defined a single musical note in terms of its pitch, duration and start time. In symbolic musical data formats, such as MIDI (Musical Instrument Digital Interface), the pitch of a note is often encoded as a discrete value that can take one of 128 values:

$$\forall i \in \{1, 2, \ldots N\} : p_i \in \{1, 2, \ldots 128\}. \tag{5.7}$$

This is enough to e.g. span the keyboard of a piano that has 88 keys. If the pitch value is increased or decreased by 1, the tone frequency is, respectively, raised or lowered by one semitone, the smallest harmonic interval used in most Western music. A numerical value, however, is not an appropriate representation to model the pitch, for the difference between

two pitch values does not necessarily reflect their harmonic relationship. Consider for example the triad chords C-E-G ('C') and C-E-G# ('C+'); their Euclidean distance is 1, but harmonically they are very different and in a composition the first can only seldom be replaced by the second, and vice versa. To counter this problem, we represent the pitch of a note by a one-hot encoding, i.e. an all-zeros vector except for a single one at the position of the numerical pitch value. This way, we do not impose an ordered structure on the pitches, so that the appropriate harmonic relationships can be learned by the generative model. The one-hot representation of p_i is denoted by $\mathbf{p}_i$.

In most Western music the duration of a note can be defined as integer multiple of the shortest note in a piece, e.g. a $1/16^{\text{th}}$ or $1/32^{\text{nd}}$ note. Such a representation, however, is not flexible and is agnostic of tempo changes. We will therefore represent both the duration of a note and its start time as an absolute real value in seconds. The start time of a note will be encoded relative to the start time of the previous note; otherwise the start time would continuously increase throughout the piece, thereby potentially impeding proper generalization of the model. For example in a chord, the first note will have a positive relative start time, and all subsequent notes will have a relative start time of zero. The representation of a note's duration d_i is denoted by $\mathbf{d}_i$, and the relative start time of a note by $\mathbf{s}_i$. We also apply a logarithmic transformation to both, since short durations and start times are more common than longer ones. We therefore arrive at the following representations:

$$\forall i \in \{1, 2, \dots N\} \colon \mathbf{d}_i = [\log(d_i + 1)]\,, \tag{5.8}$$

$$\forall i \in \{1, 2, \dots N\} \colon \mathbf{s}_i = [\log(s_i - s_{i-1} + 1)]\,,$$

$$s_0 = s_1. \tag{5.9}$$

An arbitrary note a_i is now represented by $\mathbf{a}_i$ as a concatenation of the representations for its pitch, duration and start time.

$$\forall i \in \{1, 2, \dots N\} \colon \mathbf{a}_i = \mathbf{p}_i \oplus \mathbf{d}_i \oplus \mathbf{s}_i. \tag{5.10}$$

We have used the $\oplus$ operator to denote vector concatenation. The resulting vector representation has 130 dimensions, i.e. 128 for the pitch, and single dimension for both duration and start time.

5.3.2 Modeling music with recurrent neural networks

To arrive at the factorization of Equation (5.5), we need to be able to model $P(a_i|a_{1:i-1})$. That is, we need an appropriate machine learning model that

is able to predict the next note based on the entire history of the piece. For this we will use recurrent neural networks (RNNs). These types of neural networks are especially well-suited to model time series. In this context, RNNs take a new input token $\mathbf{x}$ at every time step and produce a corresponding output, whilst maintaining an internal (hidden) representation $\mathbf{h}$ as a summary of the past series [24]:

$$\begin{aligned} \mathbf{h}_i &= f(\mathbf{x}_i, \mathbf{h}_{i-1}), \\ \mathbf{x}_{i+1} &\sim g(\mathbf{h}_i). \end{aligned} \tag{5.11}$$

In this, $f(\cdot)$ and $g(\cdot)$ are both non-linear, parameterized and differentiable functions. While $f(\cdot)$ is a deterministic function, $g(\cdot)$ outputs a probability distribution over all possible tokens, from which the next token can be sampled.

In order to solve a certain task, a suitable loss function is defined on the output of the RNN. Then, given enough example data, the RNN parameters are learned through a procedure called *backpropagation through time* [25]. In the context of modeling sequences of musical notes, the analogy between Equations (5.4) and (5.11) is clear if we regard the hidden state of the RNN as a complete representation of the past sequence. Indeed, if the initial hidden state of the RNN is kept fixed, we have that:

$$\begin{aligned} P(\mathbf{x}_{i+1}|\mathbf{h}_i) &= P(\mathbf{x}_{i+1}|\mathbf{x}_i, \mathbf{h}_{i-1}) \\ &= P(\mathbf{x}_{i+1}|\mathbf{x}_i, \mathbf{x}_{i-1}, \mathbf{h}_{i-2}) \\ &= \cdots = P(\mathbf{x}_{i+1}|\mathbf{x}_{1:i}). \end{aligned} \tag{5.12}$$

RNNs are therefore fit candidates to model polyphonic music as outlined in Section 5.2.

Suppose now that we have an RNN architecture $\mathcal{R}$, for which the output is computed through the function $\mathcal{R}(\cdot)$. The result of $\mathcal{R}(\mathbf{a}_i)$ on the i^{th} note representation is then a prediction $\hat{\mathbf{a}}_{i+1}$ for the next note. To train the RNN, we define a loss function comparing this prediction to the ground truth $\mathbf{a}_{i+1}$. Since the pitch of a note is represented by a discrete one-hot encoding, the first part $\mathcal{L}_{CE}$ of the loss function is focused on correctly classifying this pitch, for which we use a standard multiclass cross-entropy loss:

$$\mathcal{L}_{CE}\left(\hat{\mathbf{a}}_i, \mathbf{a}_i\right) = -\sum_{j=1}^{128} \mathbf{p}_{i,j} \log\left(\hat{\mathbf{p}}_{i,j}\right). \tag{5.13}$$

The duration and start time of a note, on the other hand, are represented as continuous values. For these quantities we therefore define the following ℓ_2 loss commonly used in regression problems:

$$\mathcal{L}_{\ell_2}\left(\hat{\mathbf{a}}_i, \mathbf{a}_i\right) = \left(\mathbf{d}_i - \hat{\mathbf{d}}_i\right)^2 + \left(\mathbf{s}_i - \hat{\mathbf{s}}_i\right)^2. \tag{5.14}$$

Table 5.1: The RNN architecture used for unconditioned music generation.

Layer #	Layer type (# dimensions) and nonlinearity
	Input (130)
1–2	GRU (256)
	sigmoid (gates); tanh (hidden state update)
	Dropout, $p = 0.25$
3	Fully connected dense (512)
	leaky ReLU (leakiness $= 0.01$)
	Dropout, $p = 0.25$
4	Fully connected dense (130)
	softmax with temperature τ on $\text{output}_{1:128}$
	ReLU on $\text{output}_{128:130}$

To jointly optimize both problems, the loss function that we use to train the RNN is a linear combination of both the cross-entropy and ℓ_2 loss:

$$\mathcal{L}(\hat{\mathbf{a}}_i, \mathbf{a}_i) = \mu \mathcal{L}_{CE}(\hat{\mathbf{a}}_i, \mathbf{a}_i) + (1 - \mu)\mathcal{L}_{\ell_2}(\hat{\mathbf{a}}_i, \mathbf{a}_i), \tag{5.15}$$

for a certain parameter $\mu \in [0, 1]$. Varying this parameter shifts focus to either one of the losses—i.e. pitch or rhythm—during training.

To condition the music generation on a specific composer we add a one-hot composer encoding $\mathbf{c}_i$ to the representation of each note $\mathbf{a}_i$ of a piece:

$$\forall i\colon \mathbf{a}_i = [\mathbf{p}_i, \mathbf{d}_i, \mathbf{s}_i, \mathbf{c}_i]\,. \tag{5.16}$$

Through this modification and given enough degrees of freedom, the internal dynamics of the RNN become partly dependent on the given composer and the output of $\mathcal{R}(\cdot)$ can therefore change if we switch from one composer to another. The loss function to train this particular RNN remains the same as given in Equation (5.15). In this paper we consider four different composers: Bach, Haydn, Beethoven and Chopin; the one-hot composer vectors therefore have a dimensionality of 4.

5.3.3 RNN architecture

The RNN architecture $\mathcal{R}$ for unconditioned music modeling, is given in Table 5.1. As was shown in Equation (5.10), the input as well as the output of the network is a 130-dimensional note representation. This input is fed to two stacked recurrent layers, for which we use gated recurrent units (GRU) [15]. We empirically found that using GRUs instead of the more widely used LSTMs increased training stability as well as sample quality. Thoroughly investigating the difference between LSTMs and GRUs is

Table 5.2: The RNN architecture used for music generation conditioned on composer information.

Layer #	Layer type (# dimensions) and nonlinearity
	Composer input (4)
1	Note input (130) $\oplus$ Fully connected dense (64)
2–3	GRU (256)
	sigmoid (gates); tanh (hidden state update)
	Dropout, $p = 0.25$
4	Fully connected dense (512)
	leaky ReLU (leakiness $= 0.01$)
	Dropout, $p = 0.25$
5	Fully connected dense (130)
	softmax with temperature τ on $\text{output}_{1:128}$
	ReLU on $\text{output}_{128:130}$

however beyond the scope of this paper. The output of the last GRU layer is passed through two dense layers. At the input of each dense layer we apply a dropout regularization scheme with a probability of 0.25 [26]. At the output of the network the last two dimensions, representing resp. duration and start time, are left untouched to model unbounded, real-valued quantities. We apply a softmax function with temperature parameter τ to the first 128 dimensions that represent the pitch of the note. For an n-dimensional vector $\mathbf{x}$, the i^{th} component of this particular non-linearity is defined as follows:

$$\text{softmax}(\mathbf{x}, \tau)_i = \frac{\exp\left(x_i/\tau\right)}{\sum_{j=1}^{n} \exp\left(x_j/\tau\right)}. \tag{5.17}$$

This way we obtain a normalized probability distribution over all possible pitches, and the temperature controls the shape of this distribution. For high temperatures, the distribution becomes more uniform, while for low temperatures approaching zero, it resembles the delta function centered at the output with the highest probability. The final two output dimensions representing the duration and start time are fed through a standard ReLU nonlinearity, since they can never be negative.

Table 5.2 shows the architecture that is used to condition the music generation process on a particular composer. In this, the 4-dimensional one-hot composer encoding is first fed through a dense layer. This dense layer will act as a lookup table that associates a dense 64-dimensional vector with each composer. That is, each composer is projected in a 64-dimensional continuous embedding space. Therefore, any arbitrary dis-

tribution of composer identities will result in a convex combination of the composer vectors in this embedding space. For example, the embedding vector associated with 50% Bach and 50% Chopin is located halfway on the line segment between the two composer embeddings. This way we can generate music in a style that has properties from both composers. The composer embedding is then concatenated (shown by the $\oplus$ operator) with the note representation. By feeding the composer information directly to the GRU cells, the recurrent dynamics of the model are possibly altered if we switch between composers, and we can thus generate music in a specific composer style. The rest of the architecture is similar to the unconditioned model.

5.3.4 Training and sampling details

To optimize the RNN models we outlined above, we use an approximation of the truncated backpropagation through time (BPTT) training algorithm [25]. Every training instance will be a sequence of $M+1$ notes $(\mathbf{a}_i)_{i=1}^{M+1}$ of which the first M are subsequently fed to the RNN and the last M are used as ground truth labels. Such an approach is often called teacher forcing, since we always use data from the ground truth to generate predictions. At every time step we aggregate the loss that is calculated at the output of the RNN:

$$\mathcal{L}_{agg}\big((\hat{\mathbf{a}}_i),(\mathbf{a}_i)\big) = \frac{1}{M}\sum_{i=2}^{M+1}\mathcal{L}\big(\hat{\mathbf{a}}_i,\mathbf{a}_i\big). \tag{5.18}$$

This aggregated loss is used to optimize the RNN parameters, for which we use Adam with learning rate η [27]. The RNN parameters contain the weights of each layer in the network, but we also include the initial hidden state vectors $\mathbf{h}_0$ from each recurrent layer. This way the RNN can be optimally initialized for every new input sequence. In practice, multiple sequences are grouped together during training in order to perform minibatch optimization.

To generate new music given a trained RNN model, we refer to Algorithm 5.1. In this, each new note in the composition is sampled given the previous M notes. That is, we take the sequence $(\mathbf{a}_i)_{i=1}^{M}$ of the last M notes that have been generated, and we apply this sequence at the input of the RNN. This RNN has first been re-initialized with its pretrained hidden state. The final output $\hat{\mathbf{a}}_{M+1}$ of the RNN is then taken as the next sample, and is appended to the generated sequence. This process can essentially continue forever and allows for an infinite stream of music. Since note durations and start times are modeled as continuous quantities, we quantize

Algorithm 5.1: Sampling procedure for composer-conditioned music generation

input : RNN $\mathcal{R}$, initial hidden state vector $\mathbf{h}_0$, seed sequence s, sampling length L, composer vector sequence $(\mathbf{c}_i)_{i=1}^{L}$
parameters: seed length M, temperature τ

foreach $i \in (1..L)$ **do**
  `initialize(`$\mathcal{R}$, $\mathbf{h}_0$`) // Initialize the hidden state of the RNN`
  $s' \leftarrow$ `get_last_k_notes(`$s, k = M$`)`
  foreach *note* $\mathbf{a} \in s'$ **do**
    `/* Get the RNN output for each input note concatenated with the composer information, given a temperature` τ `*/`
    $\hat{\mathbf{a}} \leftarrow \mathcal{R}(\mathbf{a} \oplus \mathbf{c}_i, \tau)$
  `/* Sample a pitch from the last RNN output and quantize the time values; add the resulting note to the sampled sequence */`
  $s \leftarrow s \oplus$ `quantize(sample(`$\hat{\mathbf{a}}$`))`

the generated notes to the nearest $1/16^{\text{th}}$ timestep and duration to arrive at rhythmically steady music.

To kick off the music generation process, we feed a seed sequence of M notes, which is manually crafted or can come from an existing composition. In order to condition the generated music on the composer identity, we supply an additional composer encoding for every input note, as given by Equation (5.16). This is also shown in Algorithm 5.1. If the next note in a piece should generated in the style of e.g. Bach, we apply the last M notes of the piece to the input of the RNN, and to each note we append the composer encoding for Bach. This way, the last M notes are interpreted and processed by the RNN as if they were written by Bach, and the next note can then be generated in the same style.

5.4 Experiments

Now that we have covered all theoretical details, we will conduct a series of experiments to assess the performance of our models.

Table 5.3: Information regarding the collected MIDI dataset.

Composer	Number of pieces	Number of notes
Bach	277	354,315
Haydn	140	348,294
Beethoven	110	464,252
Chopin	178	379,873
Total	705	1,546,734

5.4.1 Data gathering

To train the RNN models we described in the previous section, we gather a dataset of polyphonic music. In particular we will focus on piano music, since there exists extensive musical literature for this instrument across different style periods. Since we restrict ourselves to one instrument, we don't need to model instrument types, thereby reducing the complexity of the problem. We pick four composers that all have contributed heavily to the piano repertoire: Bach, Haydn, Beethoven and Chopin. The music of these composers is characteristic for different style periods in Western classical music—baroque, classicism and romanticism—thereby allowing us to model a wide variety of musical styles.

On the Classical Archives platform[1] we collect all unique piano works from the aforementioned composers in MIDI format. After filtering out duplicate compositions, we arrive at a dataset of which the details are given in Table 5.3. Notice that, although we have fewer compositions by Beethoven, the number of notes is substantially larger than for the other composers. We randomly reserve 85% of all pieces per composer to create a training set, 10% for a test set and 5% for a development set.

5.4.2 Practical and experimental settings

We will use a mini-batched version of Adam with learning rate $\eta = 10^{-4}$ during training. For this purpose, we set parameter M—introduced in Section 5.3.4—to 100 and we use a batch size of 64 note sequences. These sequences are selected as follows: we randomly select a piece from the training set, from which we randomly pick a sequence of $M + 1$ subsequent notes. This process is repeated 64 times.

The loss function defined in Equation (5.15) consists of two parts weighted by a parameter μ. In practice we find that setting μ to 0.5 yields

[1] www.classicalarchives.com

good results. We also observe that it is beneficial to train the RNN models until the train error converges to a minimum, as opposed to reducing overfitting on the development set. In that case, the model will have remembered some of the more typical phrases and harmonies 'by heart', up to a certain degree. Doing so leads to more coherent, realistically sounding and pleasing music.

The temperature parameter τ is an extra hyperparameter that needs to be tuned. During training, its value is always set to 1.0, but during sampling we set the temperature parameter τ separately for each of the composers to arrive at musically pleasing results. A too high value results in highly chaotic movements, while setting it too low leads to small cells of notes which are repeated endlessly. There is no real science in setting τ to the most appropriate value, apart from listening to the resulting music and making subjective assessments on the quality of the fragments. For Bach, Beethoven and Haydn we set $\tau = 0.7$, but for Chopin we needed a lower value of $\tau = 0.5$. This is probably because the music of Chopin is rhythmically and harmonically more complex compared to the other composers.

In Figure 5.2 we give an example of music generation for different composer styles. The sheet music is created directly from MIDI data using Sibelius[2] and is manually cleaned to enhance visual appearance. We have used a seed of two bars taken from Bach's French Suite No. 2, Allemande. This piece is part of the test set and has therefore not been used to train the model. The top system in the figure shows the original piece. In the middle we show how the seed is used to generate four bars of new music in the style of Bach. The bottom system shows the same seed sequence by Bach, after which four bars of Chopin music are generated. We can visually already determine that the Chopin fragment is stylistically very different from the other two fragments. The generated Bach music, on the other hand, appears much closer to the original composition in terms of style.

5.4.3 Network layer analysis

In order to analyze how the network has learned to distinguish between different composers, we will look at some of the hidden representations within the neural network. First, the 512-dimensional output of the second-to-last layer is investigated, which is the last hidden representation before the network output. For each of the four composers, we uniformly sample 250 note sequences of length 100 from the test set. These sequences are fed through the RNN, and the hidden representation is captured. We thus arrive at 1,000 of these representations that we project in a

[2] www.sibelius.com

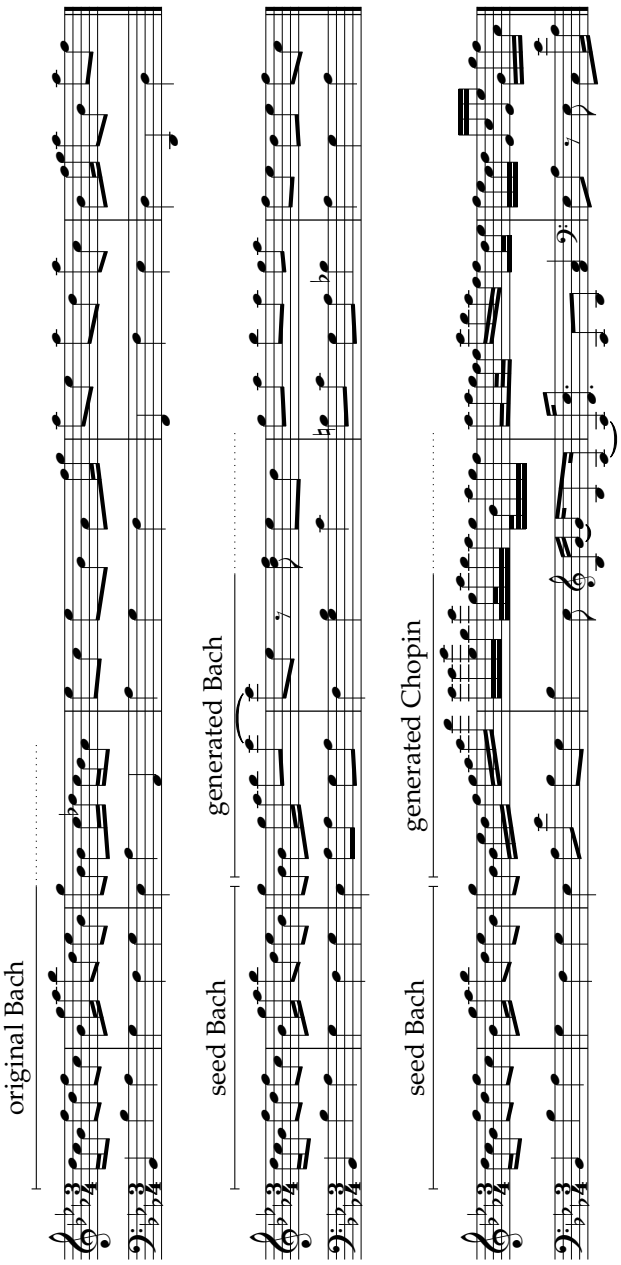


Figure 5.2: Illustration of music generation with different composer styles. On top is the original excerpt by Bach from French Suite No. 2, Allemande, BWV 813. In the middle we show music generation in the style of Bach, and at the bottom in the style of Chopin. The seed is shared between all examples.

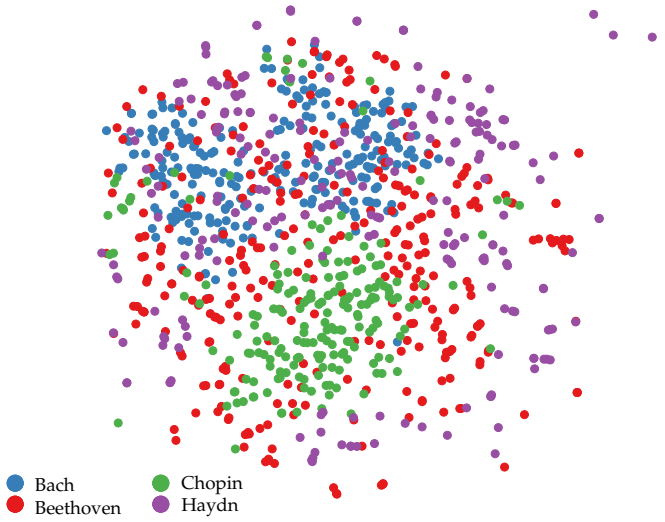


Figure 5.3: Two-dimensional t-SNE plot of 1,000 hidden representations (i.e. 250 for each of the four composers) at the second last layer in the neural network.

2D plane using t-SNE, a dimensionality reduction method focused on data visualization in two or three dimensions [28]. In this, similar data points are displayed close to each other, while dissimilar data points are typically far away from each other. The resulting plot is shown in Figure 5.3. The Bach and Chopin points are seemingly organized in different clusters. This suggest that the model should be capable of separating the style of these two composers, which is audibly fairly different. Points for Beethoven and Haydn on the other hand are scattered across the plot, and there is no clear pattern to be discovered, apart from the fact that Haydn pieces are rarely present within the Chopin cluster. From this experiment we can conclude that the network has learned, up to a certain extent, to make a distinction between the different composers, and that this distinction is reflected in the output of the neural network.

In a second analysis our focus lies on the learned composer embeddings. The embedding matrix should ideally contain an appropriate semantic representation for each composer. To investigate this, we construct a correlation matrix by calculating the Pearson correlation between each of the four composer embedding vectors, shown in Figure 5.4. We see that the Bach and Chopin embeddings are the least correlated, which is an observation similar to Figure 5.3. Beethoven and Haydn are correlated most,

	Bach	Haydn	Beethoven	Chopin
Bach	1.00	0.23	0.23	0.13
Haydn	0.23	1.00	0.38	0.24
Beethoven	0.23	0.38	1.00	0.29
Chopin	0.13	0.24	0.29	1.00

Figure 5.4: Matrix representing the Pearson correlation between each of the four composer embeddings.

which is not surprising since the early style of Beethoven is very similar to Haydn's. So, similar to the hidden representation in the previous experiment, the embeddings are a certain reflection of the relatedness between the composers.

5.4.4 User listening experiments

A lot of research has gone into the evaluation of intelligent creative systems, such as music generators, and it is not our purpose to give a complete overview on this subject. For example in 2006, Moffat et al. found that there is a discrepancy between how non-musicians and musicians evaluate generated vs human-composed music in a set-up Turing-like test [29]. Musicians seemed to be guessing at near-chance level, while non-musicians had a clear preference for human-composed music. The reason is probably that musicians have a lot of knowledge, have been analyzing music for years, and they may be acquainted with computer-generated music. They are also more likely to be familiar with the modern and 'aleatoric' genre of classical music. However, their test public only consisted of 10 non-musicians and 10 musicians, which is too low to draw accurate and statistically significant conclusions. Agres et al. pointed out that such tests, in which people are asked to label a piece of music as either human-composed or machine-

generated, are actually Consensual Assessments Techniques (CATs) rather than Turing tests in the strict sense [3]. The authors argue that subjective measures—such as Likert or continuous rating scales—are actually able to "provide very robust and consistent measures of participant judgments".

In the light of this last statement, we have conducted two experiments in a target audience of 51 persons with a solid background in classical music (but not necessarily the piano repertoire), recruited at the Royal Conservatory, Faculty of Fine Arts of the University College Ghent. We instructed the audience to evaluate short fragments of music consisting of 100 notes. We first asked to score the fragments on a scale of 0 to 10, where 0 means 'random noise' and 10 means that 'the fragment could possibly be composed by a human', which is similar to the evaluation procedure executed by Huang et al. [19]. In the experiments we stressed that scores should be given based on the quality of the composition, and not based on audio or piano sample quality. Next to a qualitative score, we also asked to indicate which one of four the composers (Bach, Haydn, Beethoven or Chopin) is stylistically closest to the fragment at hand.

In a first experiment we trained one unconditioned RNN model exclusively with Bach data, and one RNN with Chopin data. We chose these two composers since, as we have observed in the previous experiments, they are stylistically the most different. From each model we extracted 10 fragments of 100 notes after applying an initial seed sequence taken randomly from the test set. We also include 5 real compositions from Bach as well as Chopin to be able to compare them qualitatively with the generated music. The participants of the experiment did not know that there were real compositions among the fragments.

In a second experiment we examined the composer-conditioned RNN model trained on all four composers. In this we start with a seed sequence of composer 1, after which we generate 100 notes in the style of composer 1. For the next 100 notes we linearly shift the style from composer 1 to composer 2, e.g.

$$\forall i \in \{1, 2, \ldots, 100\} : \mathbf{c}_i = [{}^{(100-i)}/_{100}, {}^{i}/_{100}], \tag{5.19}$$

for 2-dimensional composer encodings. This way, as explained in Section 5.3.3, we perform a linear interpolation in the composer embedding space. The final 100 notes are all generated in the style of composer 2. So, for every possible composer pair, we have generated 300 notes, of which the first 100 are in the style of composer 1, the last 100 are in the style of composer 2, and the middle 100 show a linear transition from composer 1 to composer 2. This leads to a total of 36 fragments of 100 notes for the four composers we consider.

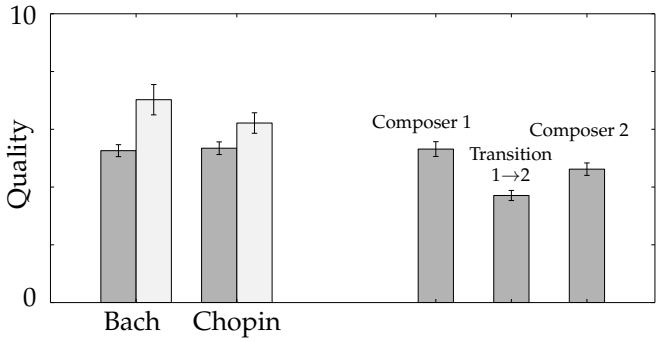


Figure 5.5: Bar chart representing composition quality. On the left we show results for experiment 1 (either Bach or Chopin), on the right for experiment 2 (a transition between two of the four composers). Dark gray is for generated music, light gray is for real compositions.

Each test subject is asked to evaluate six fragments, of which three are chosen randomly from the set of 30 unconditioned fragments, and the other three are related fragments generated as explained above. It it important to point that experiments 1 en 2 are conducted simultaneously without knowledge of the test subjects; they do not know that experiment 1 is limited to only Chopin and Bach. We also randomly shuffle the six fragments for each participant so that related fragments from the second experiment do not follow each other.

We first asked our test subjects to rate the quality of the fragments on a scale of 0 to 10, as outlined above. The results of this are shown in Figure 5.5, in which the error bars show the standard deviation. In the left part of the figure the first experiment is shown, in which we compare generated and real music from Bach and Chopin. The difference in rated quality is 1.8 points for Bach compositions, and only 0.9 points for Chopin. In a one-sided Student's t-test, the difference in rated quality for Bach is highly significant ($p = 0.002$), but this is not the case for the Chopin compositions ($p = 0.28$). These are good results, but still show that generated music cannot yet match hand-made compositions. The fact that the difference in quality for Bach is larger than for Chopin is probably explained by the fact that Bach's original style is very strict and easily recognized, and therefore it is easily noticed when a mistake or glitch is present.

The second experiment is shown in the right part of the figure. We have used all fragments for this experiment, including Haydn and Beethoven next to Bach and Chopin. The quality of the first fragment matches with the first experiment, which is as expected. During composer transition, however, the score drops approximately 1.5 points, and it climbs again 1 point for the third fragment. These observations can be due to a number of factors. For one, the network is possibly unable to adequately inter-

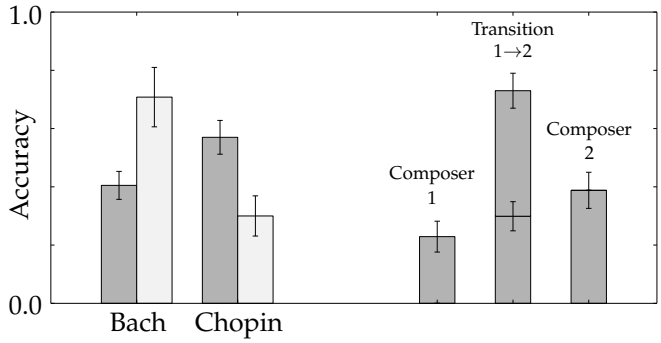


Figure 5.6: Bar chart representing composition accuracy. On the left we show results for experiment 1 (either Bach or Chopin), on the right for experiment 2 (a transition between two of the four composers). Dark gray is for generated music, light gray is for real compositions.

polate between the recurrent dynamics of different composers, so that no pleasingly sounding music results. Another possible reason is that the test subjects got confused by the style transition, and therefore gave a lower rating.

We also asked our test audience to pick a composer that is stylistically closest to each fragment they were given. Their choices are compared to the ground truth in Figure 5.6, in which we report accuracy as a performance measure. We again see a large gap of 30% for Bach compositions, again most probably because of easy identification of this composer's music. Surprisingly we see the opposite happening for Chopin, for which the generated music gets an accuracy of 57% against only 30% for the original compositions[3]; he got most often confused with Beethoven—also a romantic composer—especially Prelude No. 6, Op. 28. The difference for Bach is statistically significant, while this is not as strong for Chopin (bootstrap hypothesis test, $p = 0.002$ vs. $p = 0.10$). In the results for the second experiment we have stacked two bars for the second fragment, of which the lower bar is the accuracy for composer 1 and the top one for composer 2. The combined accuracy is almost 72.9% for the second fragment, despite the lower quality scores. The accuracy for the first fragment is only 22.9% (i.e. worse than random); the third fragment performs much better with 38.8%. However, through a bootstrap hypothesis test we cannot reject the null hypothesis that the accuracy for the first and third fragment is equal ($p = 0.19$).

[3]This reminds us of the story that Charlie Chaplin himself once lost a Charlie Chaplin impersonation contest. Among the approximately 40 competitors, he ended 27th; see http://trove.nla.gov.au/newspaper/article/70146933

5.5 Conclusion

In this paper we introduced the task of composer-conditioned music composition. We devised a recurrent neural network architecture to generate polyphonic piano music in the style of a given composer. We showed how we are able to make continuous transitions between composer styles during the music generation process. A field study was conducted in a target audience with a background in classical music. The test subjects were asked to assess the quality of generated compositions and select a composer whose style is nearest to the fragments. From the experiments we conclude that it is possible to induce composer-specific dynamics in the music generation model and that we can make transitions from one composer to another. However, generated polyphonic music is still discernible from hand-made compositions. One of the remaining question is how a global structure can be obtained, in which we have recurring themes, variations, and proper phrasings. This is currently absent in the model and the resulting compositions. Another important thought is that the teacher forcing training paradigm might not be suitable to allow for true model creativity, since we always depart from knowledge in the ground truth training data. This leaves open paths for future research.

References

[1] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. *WaveNet: A Generative Model for Raw Audio*. arXiv.org, 2016. arXiv:1609.03499v2.

[2] J. Engel, C. Resnick, A. Roberts, S. Dieleman, M. Norouzi, D. Eck, and K. Simonyan. *Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders.* ICML, 2017.

[3] K. Agres, J. Forth, and G. A. Wiggins. *Evaluation of Musical Creativity and Musical Metacreation Systems.* Computers in Entertainment, 2016.

[4] K. Ebcioglu. *An Expert System for Harmonizing Chorales in the Style of Bach,J.S.* Journal of Logic Programming, 1990.

[5] A. R. Brown, T. Gifford, and R. Davidson. *Techniques for Generative Melodies Inspired by Music Cognition*. Computer Music Journal, 2015.

[6] D. Herremans and K. Soerensen. *Composing fifth species counterpoint music with a variable neighborhood search algorithm*. Expert Systems with Applications, 2013.

[7] K. Jones. *Compositional applications of stochastic processes*. Computer Music Journal, 1981.

[8] J. F. Paiement, S. Bengio, and D. Eck. *Probabilistic models for melodic prediction*. Artificial Intelligence, 2009.

[9] D. Ponsford, G. Wiggins, and C. Mellish. *Statistical learning of harmonic movement*. Journal of New Music Research, 1999.

[10] W. Schulze and B. van der Merwe. *Music Generation with Markov Models*. Ieee Multimedia, 2011.

[11] F. Pachet and P. Roy. *Non-Conformant Harmonization - the Real Book in the Style of Take 6.* ICCC, 2014.

[12] P. M. Todd. *A connectionist approach to algorithmic composition*. Computer Music Journal, 1989.

[13] M. C. Mozer. *Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing*. Connection Science, 1994.

[14] S. Hochreiter and J. Schmidhuber. *Long short-term memory*. Neural Computation, 1997.

[15] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.* arXiv.org, 2014. arXiv:1412.3555v1.

[16] D. Eck and J. Schmidhuber. *Finding temporal structure in music: Blues improvisation with LSTM recurrent networks*. In IEEE Workshop on Neural Networks for Signal Processing, 2002.

[17] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent. *Modeling Temporal Dependencies in High-Dimensional Sequences - Application to Polyphonic Music Generation and Transcription.* ICML, 2012.

[18] I.-T. Liu and B. Ramakrishnan. *Bach in 2014: Music Composition with Recurrent Neural Network.* arXiv.org, 2014. arXiv:1412.3191.

[19] A. Huang and R. Wu. *Deep Learning for Music*. arXiv.org, 2016. arXiv:1606.04930v1.

[20] N. Jaques, S. Gu, R. E. Turner, and D. Eck. *Generating Music by Fine-Tuning Recurrent Neural Networks with Reinforcement Learning*. In Deep Reinforcement Learning Workshop, NIPS, 2016.

[21] K. Choi, G. Fazekas, and M. Sandler. *Text-based LSTM networks for Automatic Music Composition*. arXiv.org, 2016. arXiv:1604.05358v1.

[22] C. Walder. *Modelling Symbolic Music - Beyond the Piano Roll.* ACML, 2016.

[23] C. Malleson, J. C. Bazin, O. Wang, D. Bradley, T. Beeler, A. Hilton, and A. Sorkine-Hornung. *FaceDirector - Continuous Control of Facial Performance in Video.* ICCV, 2015.

[24] I. Sutskever. *Training recurrent neural networks*. PhD thesis, 2013.

[25] J. L. Elman. *Finding Structure in Time*. Cognitive science, 1990.

[26] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. *Dropout - a simple way to prevent neural networks from overfitting.* Journal of Machine Learning Research, 2014.

[27] D. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. In ICLR, 2015.

[28] L. van der Maaten and G. Hinton. *Visualizing Data using t-SNE*. Journal of Machine Learning Research, 2008.

[29] D. Moffat and M. Kelly. *An investigation into peopleś bias against computational creativity in music composition*. 2006. arXiv:11008750380463336829related:fa126CTwxpgJ.

6

Large-Scale User Modeling with Recurrent Neural Networks for Music Discovery on Multiple Time Scales

"Just take a look at the menu, we give you rock a la carte. We'll breakfast at Tiffany's, we'll sing to you in Japanese. We're only here to entertain you."

—Queen, 1978

In the final chapter of this thesis before the conclusion, we will explore how recurrent neural networks can be used in the context of recommender systems. For this purpose, we will examine the listening behavior of users on Spotify, globally the most widely used music streaming platform. On one hand, word2vec will be used to represent 6 million tracks in a low-dimensional space. This space is learned by treating user-created playlists as documents, and tracks in these playlists as individual words. On the other hand, a recurrent neural network will be trained to process the sequence of tracks that a user has listened to, and the resulting user embedding is projected in the same word2vec space. This user embedding is then used to predict a track the user will likely listen to in the future.

⋆ ⋆ ⋆

C. De Boom, R. Agrawal, S. Hansen, E. Kumar, R. Yon, C.-W. Chen, T. Demeester, and B. Dhoedt

Appeared in Multimedia Tools and Applications, online, 2017.

Abstract The amount of content on online music streaming platforms is immense, and most users only access a tiny fraction of this content. Recommender systems are the application of choice to open up the collection to these users. Standard collaborative filtering has the disadvantage that it relies on explicit consumer signals, which are often unavailable, and generally disregards the temporal nature of music consumption. On the other hand, item co-occurrence algorithms, such as the recently introduced word2vec-based recommenders, are typically left without an effective user representation. In this paper, we present a new approach to model users through recurrent neural networks by sequentially processing consumed items, represented by any type of embeddings and other context features. This way we obtain semantically rich user representations, which capture a user's musical taste over time. Our experimental analysis on large-scale user data shows that our model can be used to predict future songs a user will likely listen to, both in the short and long term.

6.1 Introduction

Online digital content providers, such as media streaming services and e-commerce websites, usually have immense catalogs of items. To prevent users from having to search manually through the entire catalog, recommender systems help to filter out items users might like to watch, listen to, buy... and are often based on characteristics of both users and items. One type of recommendation algorithms is collaborative filtering, which is generally based on item consumption signals or rating provided by users. However, such explicit information is not always available. For example, in what way does clicking on an item represent how much the user likes this item? Implicit feedback models are therefore used here, but they require careful parameter tuning. Next to this, systems that model users based on aggregate historical consumption will often ignore the notion of sequentiality. In the case of music consumption, for example, it has been investigated that a user's listening behavior can be described by a trajectory in time along different artists and genres with periods of fixations and transitions [1].

Recently, recommenders have also been built on top of item embeddings [2, 3]. Such embeddings, or vector representations, are generally

learned using item co-occurrence measures inspired by recent advances in language modeling, e.g. word2vec and related algorithms [4]. The problem with this approach is that we are left without an adequate user representation, and the question remains how to derive such a representation based on the given item embeddings.

In this work we focus on creating user representations in the context of new music discovery on online streaming platforms. We start from given latent embeddings of the items in the catalog and we represent users as a function of their item consumption. For this, we propose that a user's listening behavior is a sequential process and is therefore governed by an underlying time-based model. For example, think of a user listening to an artist's album for some time and then transitioning to the next album, or to a compilation playlist of the same musical genre. To model the dynamics of a user's listening pattern we use recurrent neural networks, which are currently among the state-of-the-art in many data series processing tasks, such as natural language [5, 6] and speech processing [7]. We do not presuppose any of the item embedding properties, such that our model is generally applicable to any type of item representation. In the next section we will explain the problem setting and highlight related work regarding music recommendation and deep learning. In Section 6.3 we will describe our methodology, after which we perform an initial data analysis in Section 6.4. We conclude with the results of our experiments in Section 6.5. A complete table with the used symbols in this article is given in Table 6.1.

6.2 Motivation and Related Work

Ever since the launch of the Netflix Prize competition in 2006 [8], research in recommender systems, and a particular subset called collaborative filtering, has spiked. The basis of modern collaborative filtering lies in latent-factor models, such as matrix factorization [9]. Herein, a low-dimensional latent vector is learned for each item and user based on rating similarities, and in the most basic scheme the dot product between a user vector $\mathbf{v}_u$ and item vector $\mathbf{v}_i$ is learned to represent the rating r_{ui} of item i by user u:

$$r_{ui} = \mathbf{v}_u^T \mathbf{v}_i. \tag{6.1}$$

This setting is based on the entire user and item history, and does not take into account that a user's taste might shift over time. Koren et al. [9] mention that such temporal effects can easily be brought into the model by adding time-dependent biases for each item and user:

$$r_{ui}(t) = \mu + b_u(t) + b_i(t) + \mathbf{v}_u^T \mathbf{v}_i(t). \tag{6.2}$$

Table 6.1: List of used symbols, in order of appearance.

$\mathbf{v}_u(t)$	User vector for user u at time t
$\mathbf{v}_i(t)$	Item vector for item i at time t
r_{ui}	rating of item i by user u
μ	Global average rating
$b_u(t)$	Rating bias of user u at time t
$b_i(t)$	Rating bias of item i at time t
h_t	Hidden state at time t
c_t	Cell state at time t
f_t	Forget gate at time t
o_t	Output gate at time t
r_t	Reset gate at time t
u_t	Update gate at time t
U_x, W_x	Weight matrices for gate x
w_x	Weight vector for gate x
b_x	Bias for gate x
$\mathcal{F}(\cdot)$	Non-linear function
$\sigma(\cdot)$	Sigmoid function
$\odot$	Element-wise multiplication operator
N	Number of songs in the catalog
D	Embedding dimensionality
U	Set of all users on the platform
$(\mathbf{s}^u)$	Ordered sequence of song vectors user u listened to
$\mathbf{t}^u$	Taste vector of user u
$\mathcal{R}(\cdot; \mathbf{W})$	RNN function with parameters $\mathbf{W}$
$\mathcal{L}(\cdot)$	Loss function
$\|\cdot\|_2$	L2 norm
$L_{\cos}(\cdot)$	Cosine distance
unif $\{x, y\}$	Uniform distribution between x and y
$\mathcal{D}$	Dataset of song sequences
$\ell_{\min}, \ell_{\max}$	Minimum and maximum sampling offsets
η	Learning rate
$\mathbf{c}^u$	Context vector for user u
$\oplus$	Vector concatenation operator
C	Ordered set of contexts on the Spotify platform
C_i	i'th context in C
$c(s)$	set of contexts for song s
onehot(i, L)	One-hot vector of length L with a 1 at position i
$\mathbf{1}_A(x)$	Indicator function: 1 if $x \in A$, else 0
$\Delta(x, y)$	Time difference between playing songs x and y
D_{hid}	Hidden dimensionality
γ	Discount factor
$\mathcal{W}(\cdot; \mathbf{w})$	Weight-based model function with weights $\mathbf{w}$
λ	Regularization term
$\zeta(\cdot)$	Riemann zeta function
Zipf$_z(\cdot)$	Zipf probability density function with parameter z
rPST, rPLT	Short- and long-term playlist RNN
rHST, rHLT	Short- and long-term user listening history RNN
bWST, bWLT	Short- and long-term weight-based model

Dror et al. [10] extend on this work by introducing additional biases for albums, artists, genres and user sessions, but these biases now represent a global rather than a temporal effect of a user's preference towards an item. This way we can for example model to what extent the average rating for a specific album is higher or lower compared to other albums. Although the models of Koren et al. and Dror et al. are capable of representing a user's overall preference towards an item and how this preference shifts in time, it cannot immediately explain why a user would rate item w higher or lower after having consumed items x, y and z. That is, a user's preference can depend on what he or she has consumed in the immediate past. Basic collaborative filtering techniques do not explicitly model this sequential aspect of item consumption and the effect on what future items will be chosen.

Next to this, standard collaborative filtering and matrix factorization techniques are mostly fit for explicit feedback settings, i.e. they are based on positive as well as negative item ratings provided by the users. In more general cases where we deal with views, purchases, clicks... we only have positive feedback signals, which are binary, non-discriminative, sparse, and are inherently noisy [11]. Standard and tested techniques to factorize a sparse matrix of user-item interactions are singular value decomposition (SVD) and non-negative matrix factorization (NMF) [12, 13]. In the context of implicit feedback, however, missing values do not necessarily imply negative samples. Pan et al. [14] therefore formulate this as a so-called one class classification problem, in which observed interactions are attributed higher importance than non-observed ones through different weighting schemes, or through careful negative sampling. Hu et al. [11] on the other hand construct an implicit matrix factorization algorithm, based on the singular value decomposition of the user-item matrix, which differs from the standard algorithm by attaching higher confidence on items with a large number of interactions during optimization. Johnson [15] uses the ideas from Hu et al. and devises a probabilistic framework to model the probability of a user consuming an item using logistic functions.

The implicit-feedback models calculate global recommendations and do not exploit temporal information to decide which items the user might be interested in. Recently, Figueiredo et al. [1] have shown that users on online music streaming services follow a trajectory through the catalog, thereby focusing their attention to a particular artist for a certain time span before continuing to the next artist. For this they use a combination of different Markov models to describe the inter- ('switch') and intra-artist ('fixation') transitions. In other work, Moore et al. [16] learn user and song embeddings in a latent vector space and model playlists using a first-order

Markov model, for which they allow the user and song vectors to drift over time in the latent space.

Compared to Markov models, which inherently obey the Markov property, recent work has shown that recurrent neural networks (RNNs) are able to learn long-term data dependencies, can process variable-length time series, have great representational power, and can be learned through gradient-based optimization. They can effectively model the non-linear temporal dynamics of text, speech and audio [18, 19], so they are ideal candidates for sequential item recommendation. In a general RNN, at each time step t a new input sample x_t is taken to update the hidden state h_t:

$$h_t = \mathcal{F}(Ux_t + Wh_{t-1}), \tag{6.3}$$

in which $\mathcal{F}(\cdot)$ is a non-linear function, e.g. sigmoid $\sigma(\cdot)$, tanh or a rectifier (ReLU and variants) [20]. To counter vanishing gradients during backpropagation and to be able to learn long-term dependencies, recurrent architectures such as long short-term memories (LSTMs) and gated recurrent units (GRUs) have been proposed, both with comparable performances [21–23]. These models use a gating mechanism, e.g. an LSTM introduces input (i_t) and forget (f_t) gates that calculate how much of the input is taken in and to what extent the hidden state should be updated, and an output gate (o_t) that leaks bits of the internal cell state (c_t) to the output:

$$\begin{aligned}
i_t &= \sigma\left(U_i x_t + W_i h_{t-1} + w_i \odot c_{t-1} + b_i\right), \\
f_t &= \sigma\left(U_f x_t + W_f h_{t-1} + w_f \odot c_{t-1} + b_f\right), \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh\left(U_c x_t + W_c h_{t-1} + b_c\right), \\
o_t &= \sigma\left(U_o x_t + W_o h_{t-1} + w_o \odot c_{t-1} + b_o\right), \\
h_t &= o_t \odot \tanh\left(c_t\right),
\end{aligned} \tag{6.4}$$

in which $\odot$ is the element-wise vector multiplication. GRUs only have a reset (r_t) and update (u_t) gate, get rid of the cell state, and have less parameters overall:

$$\begin{aligned}
r_t &= \sigma\left(U_r x_t + W_r h_{t-1} + b_r\right), \\
u_t &= \sigma\left(U_u x_t + W_u h_{t-1} + b_u\right), \\
g_t &= \tanh\left(U_g x_t + r_t \odot W_g h_{t-1} + b_g\right), \\
h_t &= (1 - u_t) \odot h_{t-1} + u_t \odot g_t.
\end{aligned} \tag{6.5}$$

Very recently there have been research efforts in using RNNs for item recommendation. Hidasi et al. [24] use RNNs to recommend items by predicting the next item interaction. The authors use one-hot item encodings

as input and produce scores for every item in the catalog, on which a ranking loss is defined. The task can thus be compared to a classification problem. For millions of items, this quickly leads to scalability issues, and the authors resort to popularity-based sampling schemes to resolve this. Such models typically take a long time to converge, and special care needs to be taken not to introduce a popularity bias, since popular items will occur more frequently in the training data. The work by Tan et al. [25] is closely related to the previous approach, and they also state that making a prediction for each item in the catalog is slow and intractable for many items. Instead, low-dimensional item embeddings can be predicted at the output in a regression task, a notion we will extend on in Section 6.3.

A popular method to learn item embeddings is the word2vec suite by Mikolov et al. [4] with both Continuous Bag-of-Words and Skip-Gram variants. In this, a corpus of item lists is fed into the model, which learns distributed, low-dimensional vector embeddings for each item in the corpus. Word2vec and variants have already been applied to item recommendation, e.g. Barkan et al. [2] formulate a word2vec variant to learn item vectors in a set consumed by a user, Liang [3] devise a word2vec-based CoFactor model that unifies both matrix factorization and item embedding learning, and Ozsoy [26] learns embeddings for places visited by users on Foursquare to recommend new sites to visit. These works show that a word2vec-based recommender system can outperform traditional matrix factorization and collaborative filtering techniques on a variety of tasks. In the work by Tan et al. item embeddings are predicted and at the same time learned by the model itself, a practice that generally deteriorates the embedding quality: in the limit, the embeddings will all collapse to a degenerate, non-informative solution, since in this case the loss will be minimal. Also, they minimize the cosine distance during training, which we found to decrease performance a lot.

In the coming sections, we will train RNNs to predict songs a user might listen to in the future as a tool to model users on online music streaming platforms. For this, we will predict preexisting item embeddings—about which we will not make any assumptions in the model—and our task is therefore a regression rather than a classification problem. This approach is closely related to the work by van den Oord et al. [27] in which collaborative latent vectors are predicted based on raw audio signals, and also related to the work by Hill et al. [28] who learn to project dictionary definition representations onto existing word embeddings. Regarding sequential item recommendation, the related works by Hidasi et al. and Tan et al. mentioned above both perform item recommendation within user sessions, i.e. uninterrupted and coherent sequences of

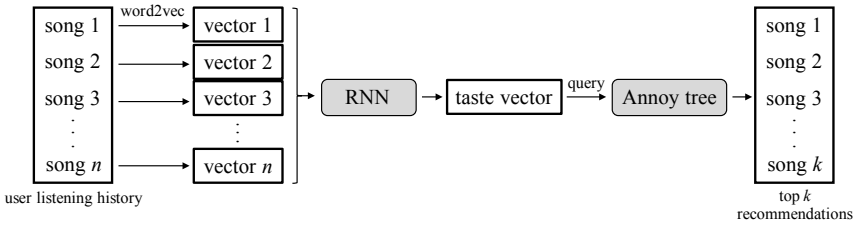


Figure 6.1: The entire song recommendation pipeline for a specific user; we start with the user's listening history of N songs, and we end the pipeline with k song recommendations.

item interactions, which can last from ca. 10 minutes to over an hour. Here, the prediction time scale is very short-term, and since consumed items within a user session are usually more similar than across user sessions, it is generally easier perform item recommendation on this short time scale. In this work we will explore recommending songs for short-term as well as long-term consumption. To recommend songs on the long term, we will need to be able to model a user's behavior across session boundaries.

6.3 RNNs for Music Discovery

In this section we will explain the details of our approach to use RNNs as a means to model users on online music streaming platforms and to recommend new songs in a music discovery context. Since we aim towards models that can be used efficiently in large-scale recommendation pipelines, we require the models to be trained within a reasonable time frame. Furthermore, sampling from the model should be efficient. Small models with little parameters are typically wanted to satisfy both requirements.

The entire recommendation pipeline for one specific user is given in Figure 6.1. The basic building blocks of the pipeline are song vectors, which have been learned using the songs in the catalog. The general idea is then to capture and predict a *taste vector* for each user. These taste vectors are the output of an RNN that sequentially aggregates song vectors from the user's listening history, and can therefore be regarded as a representation of the user's musical taste. The taste vector can subsequently be used to generate song recommendations by querying a tree data structure for the nearby song vectors.

Since we construct real-valued taste vectors, the RNN solves a regression task rather than a classification task, as argued in Section 6.2. Directly predicting item embeddings is a regression problem that requires predict-

ing a limited set of real-valued outputs, as opposed to a classifier with as many outputs as the number of items. The computational footprint of these models is typically smaller than the classifiers. They are usually learned faster, and are not per se biased towards popular items. One of the main advantages is that any type of item embeddings and embedding combinations, along with other features, can be used to learn the regression model.

We break the recommendation pipeline into three separate stages which we will cover below. First, we learn low-dimensional embeddings for each song in the streaming catalog using word2vec (§6.3.1). Then, we use an RNN to generate taste vectors for all users in the song embedding space (§6.3.2). Finally, we use the taste vector to query songs in the nearby space to generate recommendations for all users (§6.3.3).

6.3.1 Learning song embeddings

In the first stage of the recommendation pipeline we learn latent vector representations for the top N most popular songs in the catalog. For this, we use Google's word2vec suite as explained in Section 6.2; more specifically we use the Continuous Bag-of-Words (CBoW) algorithm with negative sampling. As input to the word2vec algorithm we take user-created playlists of songs. In this, each playlist is considered as an ordered 'document' of songs. By scanning all playlists in a windowed fashion, word2vec will learn a distributed vector representation $\mathbf{s}$ with dimensionality D for every song s. Details regarding training data for word2vec will be highlighted in Section 6.4.

6.3.2 Learning user taste vectors

In the second pipeline stage we use RNNs to produce user taste vectors based on song listening history. The network takes a sequence of song vectors of dimensionality D as input and produces a single taste vector with the same dimensionality D. Let's denote the set of all users by U, the ordered sequence of song vectors user u listened to by $(\mathbf{s}^u)$, and the predicted taste vector by $\mathbf{t}^u$. The RNN then produces:

$$\forall u \in U \colon \mathbf{t}^u = \mathcal{R}\left((\mathbf{s}^u); \mathbf{W}\right), \tag{6.6}$$

in which $\mathcal{R}(\cdot\,; \mathbf{W})$ represents the function the RNN computes with parameters $\mathbf{W}$.

To learn a semantically rich user taste vector that is able to generate adequate recommendations, ideally this taste vector should be able to capture how a user's listening behavior is evolving over time. We therefore train

the RNN to predict a song the user is going to listen to in the future. More specifically, for a particular user u, we take the first n consecutive songs $\mathbf{s}^u_{1:n}$ this user has listened to, and we try to predict a future song vector $\mathbf{s}^u_{n+\ell}$, for some strictly positive value of ℓ. As a loss function, we use the L_2 distance between the predicted taste vector and the true future song vector:

$$\mathcal{L}\left(\mathbf{s}^u_{n+\ell}, \mathcal{R}\left(\mathbf{s}^u_{1:n}; \mathbf{W}\right)\right) = \left\|\mathbf{s}^u_{n+\ell} - \mathcal{R}\left(\mathbf{s}^u_{1:n}; \mathbf{W}\right)\right\|_2. \tag{6.7}$$

We experimented with other distance functions, such as cosine distance and a weighted mixture of cosine and L_2 distance. The cosine distance $L_{\cos}(\cdot)$ between two vectors $\mathbf{x}$ and $\mathbf{y}$ is given by:

$$L_{\cos}\left(\mathbf{x}, \mathbf{y}\right) = 1 - \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2}. \tag{6.8}$$

The L_2 distance, nevertheless, gave the best results in the experiments.

To determine the best value of the prediction offset ℓ we consider two separate prediction tasks: short-term and long-term prediction. The idea is that it is generally easier to predict a song a user is going to listen to next—e.g. think of a user shuffling an artist album, or listening to a rock playlist—than it is to predict the 50th song he is going to listen to in the future. The underlying dynamics of a short-term and long-term prediction model will therefore be different. For example, the short-term model will intuitively be more focused on the last few tracks that were played, while the long-term model will generally look at a bigger timeframe. During training we sample a value of ℓ for every input sequence from a discrete uniform distribution. More specifically, for the short-term model, ℓ is sampled from unif$\{1, 10\}$, and ℓ is sampled from unif$\{25, 50\}$ for the long-term model. Random sampling of the prediction offset for every new input sequence reduces chances of overfitting and also increases model generalization. The entire training procedure is sketched in Algorithm 6.1. In here, we use a stochastic minibatch version of the gradient-based Adam algorithm to update the RNN weights [29]. We have also experimented with setting ℓ fixed to 1 for the short-term model, but this leads to very near-sighted models that make predictions almost solely based on the last played song. For example, first listening to 100 classical songs and then to 1 pop song would lead to pop song predictions by the RNN.

6.3.3 Recommending songs

The output of the RNN is a user taste vector that resides in the same vector space as all songs in the catalog. Since we trained the RNN to produce these taste vectors to lie close to future songs a user might play—in terms

Algorithm 6.1: RNN training procedure

input : dataset $\mathcal{D}$ of song sequences, initial RNN parameters $\mathbf{W}$
parameter: sequence length n, offsets $\ell_{\min}$ and $\ell_{\max}$, learning rate η

repeat
  `shuffle(`$\mathcal{D}$`)`
  foreach *batch* $B \in \mathcal{D}$ **do**
    loss $\leftarrow 0$
    foreach *sequence* $(\mathbf{s}) \in B$ **do**
      $\ell \sim \text{unif}\{\ell_{\min}, \ell_{\max}\}$
      loss $\leftarrow$ loss $+ \mathcal{L}\left(\mathbf{s}_{n+\ell}, \mathcal{R}\left(\mathbf{s}_{1:n}; \mathbf{W}\right)\right)$
    $\mathbf{W} \leftarrow$ `adam_update(`$\mathbf{W}$, loss, η`)`
until *convergence*

of L_2 distance—we can query for nearby songs in that vector space to generate new song recommendations. To scale the search for nearby songs in practice, we construct an Annoy[1] tree datastructure with all song vectors. The Annoy tree will iteratively divide the vector space in regions using a locality-sensitive hashing random projection technique [30], which facilitates approximate nearest neighbor querying. To generate suggested recommendations for a particular user, we query the Annoy tree using the user's taste vector to find the top k closest catalog songs in the vector space.

6.3.4 Incorporating play context

In general we do not only know the order in which a user plays particular songs, but we also know the context in which the songs have been played. By context we mean a playlist, an album page, whether the user deliberately clicked on the song, etc. This additional information can be very useful, e.g. we can imagine that a user clicking on a song is a stronger indicator of the user's taste than when the song is played automatically in an album or playlist after the previous song has finished playing. Since the RNN can process any combination of arbitrary embeddings and features, we can supply a context vector $\mathbf{c}^u$ as extra input. The context vector is in this case concatenated with the song vector at each time step to produce the user taste vector:

$$\forall u \in U\colon \mathbf{t}^u = \mathcal{R}\left(\left(\mathbf{s}^u \oplus \mathbf{c}^u\right); \mathbf{W}\right), \tag{6.9}$$

[1] github.com/spotify/annoy

in which we use the symbol $\oplus$ to denote vector concatenation. To construct a context vector we consider the ordered set $C = (\text{album}, \text{playlist}, \text{artist}, \text{click}, \dots)$ of all possible contexts, denote C_i as the i'th context in C, and $c(s)$ as the play context for a particular song s, e.g. $c(s) = \{\text{playlist}, \text{click}\}$. The context vector for a song s is then constructed using a one-hot encoding scheme:

$$\mathbf{c} = \sum_{i=1}^{|C|} \text{onehot}(i, |C|) \cdot \mathbf{1}_{c(s)}(C_i), \tag{6.10}$$

in which $\text{onehot}(i, |C|)$ is a one-hot vector of length $|C|$ with a single 1 at position i, and in which $\mathbf{1}_A(x)$ is the indicator function that evaluates to 1 if $x \in A$ and to 0 otherwise. We also include the time difference between playing the current song and the last played song. The final context vector $\mathbf{c}_j^u$ for the j'th song s_j^u played by user u then becomes:

$$\mathbf{c}_j^u = \Delta(s_j^u, s_{j-1}^u) \oplus \sum_{i=1}^{|C|} \text{onehot}(i, |C|) \cdot \mathbf{1}_{c\left(s_j^u\right)}(C_i), \tag{6.11}$$

in which $\Delta(s_j, s_{j-1})$ is the time difference in seconds between playing song s_j and s_{j-1}, evaluating to 0 if s_{j-1} does not exist.

6.3.5 User and model updates

The recommendation pipeline we described in this section can be used in both static and dynamic contexts. In dynamic contexts, the model should be updated frequently so that recommendations can immediately reflect the user's current listening behavior. This is easily done in our model: for every song the user listens to we retrieve its song vector that we use to update the hidden state of the RNN, after which we calculate a new taste vector to generate recommendations. This requires that we keep track of the current RNN states for every user in the system, which is a small overhead. In more static contexts, in which recommendations are generated on a regular basis for all users (every day, week...), there is no need to update the RNN for every song a user listens to. Here, we retrieve the entire user listening history–or the last n songs–which we feed to a newly initialized RNN. We therefore do not need to remember the RNN states for every user.

All recommendation modules that are deployed in practical environments have to be updated regularly in order to reflect changes in the item catalog and user behavior. This is no different for the framework we present here. In order to perform a full model update, we subsequently train word2vec on the playlists in the catalog, retrain the RNN model on

the new song vectors, and populate the Annoy tree with the same song vectors. Only retraining word2vec is not sufficient since we almost never end up in the same song embedding space. If we use dynamic user updates, note that we also have to do one static update after retraining word2vec and the RNN, since the remembered RNN states will have become invalid.

6.4 Data Gathering and Analysis

The first stage in the recommendation pipeline considers the learning of semantically rich song vectors, for which we use the word2vec algorithm. In this section we explain how we gather data to train this word2vec model, and we perform a preliminary analysis on the song vectors. Finally we detail the construction of the train and test data for the RNNs.

6.4.1 Training word2vec

To create training data for the word2vec model we treat user-created playlists as documents and songs within these playlists as individual words, as mentioned in Section 6.3. For this, we gather all user-created playlists on the Spotify music streaming platform. In these playlists we only consider the top N most popular tracks; the other tracks are removed. In our experiments we set N to 6 million, which makes up for most of the streams on the Spotify platform. After filtering out unpopular tracks, we further only consider playlists with a length larger than 10 and smaller than 5000. We also restrict ourselves to playlists which contain songs from at least 3 different artists and 3 different albums, to make sure there is enough variation and diversity in the playlists. After applying the filtering above, we arrive at a corpus of 276.5 million playlists in total. In the following step, the playlists are fed to the word2vec suite, where we use the CBoW algorithm with negative sampling. We go through the entire playlist corpus once during execution of the algorithm in order to produce vectors with dimensionality $D = 40$, a number that we empirically determined and produces good results.

6.4.2 Data processing and filtering

In Section 6.5 we will train different RNN versions, for which we will use both playlist and user listening data. Playlists are usually more contained regarding artists and musical genres compared to listening data. Modeling playlist song sequences with RNNs will therefore be easier, but we can

miss out important patterns that appear in actual listening data, especially if a user listens to a wide variety of genres.

For the playlist data we extract chunks of 110 consecutive songs, i.e. 60 songs to feed to the RNN and the next 50 songs are used as ground truth for the prediction. Regarding the user listening, which is captured in the first half of 2016 on the Spotify platform, we only keep songs which have a song vector associated with them; other songs are removed from the user's history. We also save context information for every played song; for this, we consider the following 13 Spotify-related contexts: *collection*, *library*, *radio*, *own playlist*, *shared playlist*, *curated playlist*, *search*, *browse*, *artist*, *album*, *chart*, *track*, *clicked*, and *unknown* for missing context information. To extract RNN training sequences we take chunks of 150 consecutive songs, for which the first 100 songs are again used as input to the RNN and the last 50 as ground truth. We allow more songs as input to the RNN compared to the playlist training data since user listening data is generally more diverse than playlist data, as mentioned before.

Since users often return to the same songs or artists over and over again, we apply additional filtering to eliminate those songs. This will greatly improve the RNN's generalization, and will counter what we call the 'easy prediction bias', as it is too easy to predict songs a user has already listened to. This filtering is not needed for playlist data, since a song only appears once in a playlist most of the times. The filtering rules we include, are:

1. The last 50 songs should be unique;
2. The last 50 songs should not appear in the first 100 songs;
3. The last 50 artists should be unique;
4. The last 50 artists should not appear in the first 100 artists.

Note that we only remove similar songs and artists from the ground truth labels in the dataset, and that we leave the first 100 songs in the user's history intact. That is, the same song can still appear multiple times in these first 100 songs, thereby steering the user's preference towards this particular song and influencing the predictions made by the RNN. For both playlist and listening data we gather 300,000 train sequences, 5,000 validation sequences and 5,000 test sequences.

6.4.3 User data analysis

To analyze the gathered data, we take all 5,000 listening history sequences in the test set, and we calculate pairwise cosine distances between the song vectors in these sequences. We measure both the vector distance between

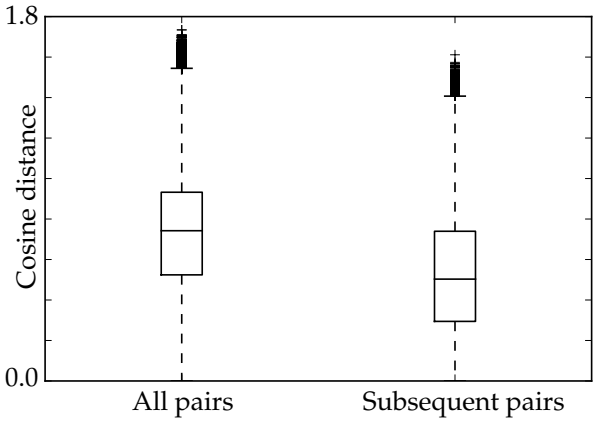


Figure 6.2: Box plots of pairwise distances between songs in all test set listening histories.

all possible song pairs in a listening sequence, as well as the distance only between subsequent songs. The distances are given as box plots in Figure 6.2, in which the whiskers are drawn at 1.5 times the interquartile range. We see that the all pairs median distance is larger than the subsequent pairs median distance by around 0.25, indeed confirming the higher correlation between subsequent songs. We also plot the number of song transitions within each user's listening history that have a cosine distance larger than 1, meaning that the next song is more different from the current song than it is similar to it. The histogram for this is shown in Figure 6.3. Most listening histories have little such song transitions. The median number is seven, which points, on average, to listening periods of 21 similar songs before transitioning to a more different region in the song space, which in turn corresponds to coherent listening sessions of about 1 to 1.5 hours long.

6.5 Experiments

In this section we discuss the results of various experiments and we show examples of song recommendations produced by different RNN models. In the following we first design the best RNN architecture by experimentation, after which we explain the baselines against which the RNN models will be tested. All experiments are run on an Amazon AWS instance, 32-core Intel Xeon 2.60GHz CPU, 64GB RAM and Nvidia GRID K520 GPU with cuDNN v4. The RNN models are implemented in Lasagne[2] and Theano [31].

[2]github.com/Lasagne/Lasagne

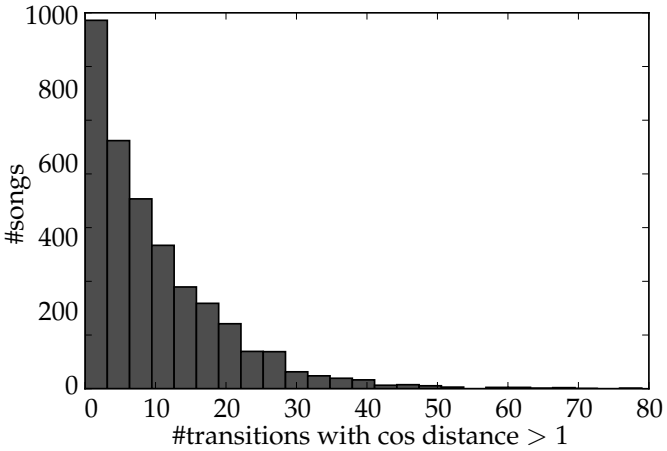


Figure 6.3: Histogram of the number of song transitions with cosine distance larger than 1.

6.5.1 Network architecture

The neural network architecture we propose consists of a number of recurrent layers followed by a number of dense layers. The number of input dimensions is $D = 40$, which is the dimensionality of a single song vector. For the recurrent layers we consider both LSTMs and GRUs, since such models are state-of-the-art in various NLP applications and can handle long-term dependencies. We use standard implementations of these layers as described in Section 6.2, with a hidden state dimensionality of all layers equal to D_{hid}. We will describe how the optimal number of recurrent layers and the optimal value of D_{hid} are chosen.

After the recurrent layers we consider two dense layers. We empirically set the dimensionality of the first dense layer to $4 \cdot D_{hid}$, and we use a leaky ReLU nonlinearity with a leakiness of 0.01 [20]. Since we are predicting a user taste vector in the same song space, the output dimensionality of the last dense layer is $D = 40$, which is the same as the input dimensionality. In the last layer we use a linear activation function, since we are predicting raw vector values.

In a first experiment we set D_{hid} to 50, we switch the recurrent layer type between LSTM and GRU, and we also vary the number of recurrent layers from 1 up to 3. We record train loss, validation loss, average gradient update time for a single batch and average prediction time for a single sequence. For this experiment we use the gathered playlist data, and we train a short-term model for which we report the best values across 15 epochs of training. The results are shown in Table 6.2. We see that both models

Table 6.2: Comparing recurrent layer type and number of layers on short-term playlist prediction.

	Train loss	Validation loss	Update [ms]	Prediction [ms]
LSTM, 1 layer	258.8	271.2	20	4.4
LSTM, 2 layers	257.3	270.2	39	8.7
LSTM, 3 layers	258.1	271.0	58	13.0
GRU, 1 layer	259.0	271.0	16	2.2
GRU, 2 layers	257.3	270.5	30	4.1
GRU, 3 layers	257.5	270.6	45	6.1

are very comparable in terms of validation loss, and that the optimal number of recurrent layers is two. The LSTM model performs slightly better than the GRU model in this case, but the GRU model predicts more than 50% faster than the LSTM model, which can be important in large-scale commercial production environments. We observe comparable results for listening history prediction as well as long-term prediction. We therefore pick the two-layer GRU model as the optimal architecture.

Next we perform experiments to determine the optimal hidden state size D_{hid}. We train the two-layer GRU architecture from above for a short-term playlist prediction, and we vary D_{hid} from 20 to 100 in steps of 10. The results are shown in Table 6.3. The validation loss is the highest at 20, and is minimal at D_{hid} values of 50 and 60. The loss remains more or less stable if we increasing the hidden state size to 100. Since larger hidden state sizes imply slower prediction and train times, we choose $D_{hid} = 50$ as the optimal hidden state size. These observations are also valid for long-term prediction and for listening history data. The final architecture is displayed in Table 6.4. It turns out that this architecture is also near-optimal for long-term playlist prediction as well as for user listening data. Since the number of network parameters is low and given the large amount of training data, we do not need additional regularization.

We also train short-term user listening RNNs with additional play context information at the input, as described in Section 6.3.4, and with the same configurations as in Tables 6.2 and 6.3. The optimal configuration is a 2-layer architecture with D_{hid} equal to 100, but the differences in performance for $50 \leq D_{hid} < 100$ are minimal. Furthermore, the performance gain compared to the same model without context information is non-significant. We will therefore disregard play context models in the rest of the experiments.

Table 6.3: Comparing performance with varying hidden layer size D_{hid} on short-term playlist prediction.

D_{hid}	L_2 validation loss	Cosine validation loss
20	287.3	0.464
30	276.4	0.425
40	271.5	0.408
50	270.5	0.406
60	270.0	0.406
70	270.4	0.406
80	270.5	0.407
90	270.1	0.405
100	270.4	0.406

6.5.2 Baselines

We will compare the performance of different RNN models against several baseline models. The general idea here is again that a user's listening behavior is governed by underlying temporal dynamics. We consider three types of baseline models: an exponential discount model, a weight-based model, and a classification model. The first two types are aggregation models, which means that they take an arbitrary number of song vectors as input and produce one output vector by mathematically combining the song vectors, e.g. through summing or taking a maximum across dimensions. Aggregation of distributed vectors is a popular practice in NLP applications and deep learning in general since it does not require any training when the vector space changes [32, 33]. In our case however, the danger of aggregation is that sometimes songs from different genres are summed together, so that we can end up in 'wrong' parts of the song space. For example, if we aggregate the song vectors of a user listening to a mix of classical and pop music, we might arrive in a space where the majority of songs is jazz, which in turn will lead to bad recommendations. The third type of baseline is based on the work by Hidasi et al. [24] as mentioned in Section 6.2. In this, we will not predict an output vector to query the Annoy LSH tree, but we will output probability scores for all N items in the catalog. The top k items can then immediately be recommended.

Exponential discount model

In the exponential discount model we make the assumption that a user's taste is better reflected by the recent songs he has listened to than by songs

Table 6.4: The final neural network architecture.

	Layer type (no. of dimensions) and non-linearity
	Input (40)
1	GRU (50)
	sigmoid (gates); tanh (hidden update)
2	GRU (50)
	sigmoid (gates); tanh (hidden update)
3	Fully connected dense (200)
	Leaky ReLU, leakiness $= 0.01$
4	Fully connected dense (40)
	Linear activation

he has listened to a while ago. We model this type of temporal dynamics by the following exponentially decaying weight model:

$$\forall u \in U\colon \mathbf{t}^u = \sum_{j=1}^{k} \mathbf{s}_j^u \cdot \gamma^{k-j}. \tag{6.12}$$

In this, we consider the song history of user u which has length k, and we weigh every vector by a power of γ, the discount factor. Setting $\gamma = 1$ results in no discounting and leads to a simple sum of the song vectors, while setting $\gamma < 1$ focuses more attention on recently played songs. The smaller γ, the more substantial this contribution becomes compared to songs played a longer time ago. If $\gamma = 0$, the user taste vector is essentially the vector of the last played song.

Weight-based model

This model is based on the weighted word embedding aggregation technique by De Boom et al. [34]. As in the exponential discount model, we will multiply each song vector $\mathbf{s}_j$ by a weight w_j:

$$\forall u \in U\colon \mathbf{t}^u = \mathcal{W}\left(\left(\mathbf{s}^u\right); \mathbf{w}\right) = \sum_{j=1}^{k} \mathbf{s}_j^u \cdot w_j, \tag{6.13}$$

in which we gather all weights w_j in a k-dimensional vector $\mathbf{w}$. Now, instead of fixing the weights to an exponential regime, we will learn the weights through the same gradient descent procedure as in Algorithm 6.1.

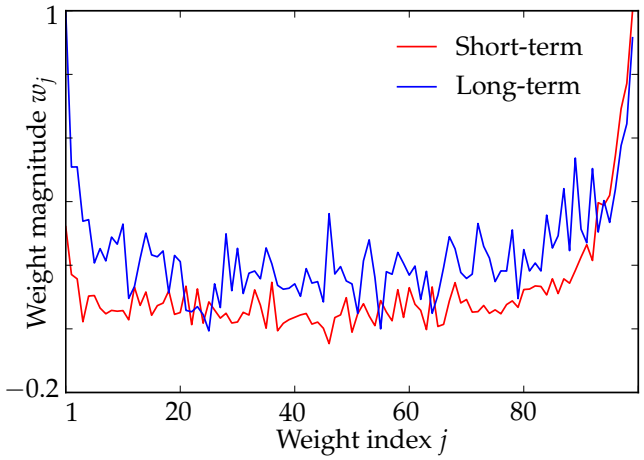


Figure 6.4: Normalized weight magnitudes for long- and short-term prediction.

In this algorithm we replace the RNN loss by the following weight-based loss:

$$\mathcal{L}\left(\mathbf{s}^u_{k+\ell}, \mathcal{W}\left(\left(\mathbf{s}^u\right); \mathbf{w}\right)\right) = \left\|\mathbf{s}^u_{k+\ell} - \mathcal{W}\left(\left(\mathbf{s}^u\right); \mathbf{w}\right)\right\|_2 + \lambda \left\|\mathbf{w}\right\|_2 . \tag{6.14}$$

We include the last term as weight regularization, and we empirically set λ to 0.001. Apart from this regularization term, we do not imply any restrictions on or relations between the weights. We train a weight-based model on user listening data for both short- and long-term predictions, and the resulting weights are plotted in Figure 6.4. For the short-term prediction the weights are largest for more recent tracks and decrease the further we go back in the past. This confirms the hypothesis that current listening behavior is largely dependent on recent listening history. For the long-term model we largely observe the same trend, but the weights are noisier and generally larger than the short-term weights. Also, the weights increase again in magnitude for the first 10 tracks in the sequence. This may signify that a portion of a user's long-term listening behavior can be explained by looking at songs, genres or artists he has listened to in the past and returns to after a while.

Classification model

As mentioned above, we loosely rely on the work by Hidasi et al. [24] to construct a classification baseline model. In this work, the items are encoded as a one-hot representation, and the output of the model is a prob-

ability score for every item in the catalog. To be able to fairly compare between all other baselines, and to help scale the model, we slightly adapt it and use the word2vec vectors as input instead of the one-hot item encodings. We employ the same neural network model as in Table 6.4 and at the output we add an additional dense layer with output dimensionality $N = 6{,}000{,}000$ and softmax activation function. The memory footprint of this softmax model thus substantially increases by around 938MiB, compared to the model in Table 6.4. Given the large output dimensionality, we also experimented with a two-stage hierarchical softmax layer [35], but the computational improvements were only marginal and the model performed worse.

We train the softmax classification model with two different loss functions. First, we consider the categorical cross-entropy loss in the case there is only one target:

$$\mathcal{L}\left(i, \mathcal{R}\left(\mathbf{s}^u_{1:k}; \mathbf{W}\right)\right) = -\text{onehot}\left(s^u_{k+\ell}, N\right) \cdot \log\left[\text{softmax}\left(\mathcal{R}\left(\mathbf{s}^u_{1:k}; \mathbf{W}\right)\right)\right]^\top. \tag{6.15}$$

In this loss function, i is the RNN output index of the target song to be predicted, and $\text{softmax}\left(\mathcal{R}\left(\mathbf{s}^u_{1:k}; \mathbf{W}\right)\right)$ is the output of the RNN after a softmax nonlinearity given the input vectors $\mathbf{s}^u_{1:k}$. The second loss function is a pairwise ranking loss used in the Bayesian Personalized Ranking (BPR) scheme by Rendle et al. [36]. This loss function evaluates the score of the positive target against randomly sampled negative targets:

$$\mathcal{L}\left(i, \mathcal{R}\left(\mathbf{s}^u_{1:k}; \mathbf{W}\right)\right) = -\frac{1}{N_S}\sum_{j=1}^{N_S} \log\left[\sigma\left(\mathcal{R}\left(\mathbf{s}^u_{1:k}; \mathbf{W}\right)_i - \mathcal{R}\left(\mathbf{s}^u_{1:k}; \mathbf{W}\right)_j\right)\right]. \tag{6.16}$$

In this, N_S is the number of negative samples that we set fixed to 100, $\sigma(\cdot)$ is the sigmoid function, and i is again the output index of the positive sample. Note that we use a sigmoid nonlinearity rather than a softmax. In practice we also add an $L2$ regularization term on the sum of the positive output value and negative sample values. To generate negative samples, we sample song IDs from a Zeta or Zipf distribution with parameter $z = 1.05$, which we checked empirically on the song unigram distribution:

$$\text{Zipf}_z(k) = \frac{k^{-z}}{\zeta(z)}, \tag{6.17}$$

in which $\zeta(\cdot)$ is the Riemann zeta function. We resample whenever a song appears in a user's listening data to make sure the sample is truly negative.

Hidasi et al. reported better stability using BPR loss compared to cross-entropy loss, but our sampling-based training procedure from Algorithm

6.1 did not produce any unstable networks for both loss functions. We trained short-term and long-term networks on the filtered listening history data using both cross-entropy and BPR, and all models took around 2.5 days to converge. By comparison, training until convergence on the same hardware only took 1.5 hours for the models presented in this work.

6.5.3 Results

In this section we will display several performance metrics on the test set of user listening histories. After all, the music recommendations will be based on what a user has listened to in the past. We have trained four RNN models: a playlist short-term (*rPST*) and long-term (*rPLT*) RNN, and a user history short-term (*rHST*) and long-term (*rHLT*) RNN. We also report metrics for five baselines: a short-term (*bWST*) and long-term (*bWLT*) weight-based model, and exponential discount models with $\gamma \in \{1.0, 0.97, 0.85\}$. In the following we will perform a forward analysis to evaluate how well a taste vector is related to future song vectors, which will show the predictive capacity of the different models. We will also do a backwards analysis to study on what part of the listening history sequences the different models tend to focus. We conclude with results on a song prediction task.

Forward analysis

In the forward analysis we take the first 100 songs of a user's listening history, which we use to generate the taste vector. This taste vector is then compared to the next 50 song vectors in the listening history in terms of cosine distance. That is, for each user u in the test set we calculate the sequence $\left(L_{\cos}(\mathbf{t}^u, \mathbf{s}^u_{100+j})\right)$, for $j \in \{1, 2, \ldots, 50\}$. Figure 6.5 shows a plot of these sequences averaged over all users in the test set. The overall trend of every model is that the cosine distance increases if we compare the taste vector to songs further in the future. This is not surprising since it is generally easier to predict nearby songs than it is to predict songs in the far future, because the former are usually more related to the last played songs. We see that the $\gamma = 0.85$ and rPST model have comparable performance. They have low cosine distance for the first few tracks, but this quickly starts to rise, and they both become the worst performing models for long-term prediction. All other models, apart from rHST and rHLT, behave similarly, with rPLT being slightly better and the $\gamma = 0.97$ model slightly worse than all others. The two best performing models are rHST and rHLT. Until future track 20, the rHST model gives the lowest cosine distance, and rHLT is significantly the best model after that. Since playlists

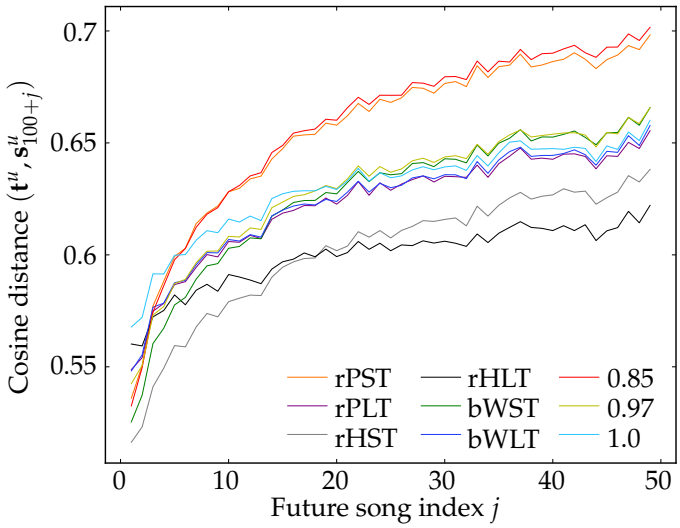


Figure 6.5: Forward analysis of the taste vector models on filtered listening history data.

are typically more coherent than listening histories—e.g. they often contain entire albums or sometimes only songs by the same artist—this can explain why the playlist-trained RNNs, and especially rPST, perform not that well in this analysis. Another general trend is that ST models typically perform better than their LT counterparts in the very near future. And at some point the LT model becomes better than the ST model and is a better predictor on the long term. Finally, among all baselines, we also observe that bWST is the best performing short-term model, and bWLT performs best to predict on the long term, which is not surprising since the weight-based models are a generalization of the discounting models. Note that the classification models remain absent, because in this case the output of the RNN is not a user taste vector.

Backwards analysis

In this analysis we again take the first 100 songs of a user' listening history, which we use to generate a taste vector. We then compare this taste vector to these first 100 songs, i.e. the songs that generated the taste vector. We thus look back in time to gain insights as to what parts of the listening history contribute most or least to the taste vector. For this we calculate the sequence $\left(L_{\cos}(\mathbf{t}^u, \mathbf{s}^u_j)\right)$, for $j \in \{1, 2, \ldots, 100\}$, and Figure 6.6 plots this sequence for each model averaged over all users in the test set. We see

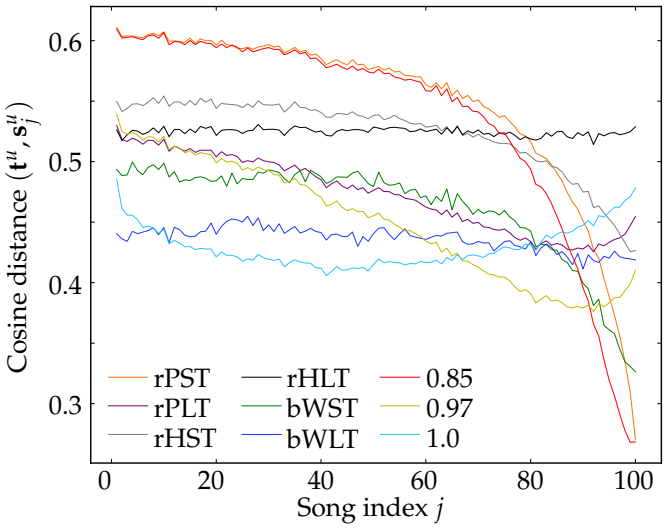


Figure 6.6: Backwards analysis of the taste vector models on filtered listening history data.

that the rPST and $\gamma = 0.85$ models are very focused on the last songs that were played, and the average cosine distance increases rapidly the further we go back in history: for songs 1 until 80 they are the worst performing. These models will typically be very near-sighted in their predictions, that is, the song recommendations will mostly be based on the last 10 played tracks. This is again due to the fact that playlists are very coherent, and predicting a near-future track can be done by looking at the last tracks alone. The rHST and bWST models also show a similar behavior, but the difference in cosine distance for tracks in the near and far history is not as large compared to rPST. The listening history RNNs, both rHST and rHLT, produce an overall high cosine distance. These models are therefore not really tied to or focused on particular songs in the user's history. It is interesting to note that the plot for rHLT and bWLT is a near-flat line, so that the produced taste vector lies equally far from all songs in terms of cosine distance. In comparison, the $\gamma = 1.0$ taste vector, which is actually just a sum of all songs, produces a U-shaped plot, which is a behavior similar to the long-term weights in Figure 6.4. If we would attribute more weight to the first and last few tracks, we would end up with a flatter line. The $\gamma = 0.97$ plot also has a U-shape, but the minimum is shifted more towards the recent listening history. Note again that the classification models are absent in this analysis for the same reason as specified above.

Precision@k

In this section we calculate the precision of actual song recommendations. We again take 100 songs from a user's history which we use to generate a taste vector. Then, as described in Section 6.3.3, we query the word2vec space for the k nearest songs in the catalog in terms of cosine distance. We will denote the resulting set as $\Omega\left(\mathbf{t}^u, k\right)$. The *precision@k* value is then the fraction of how many songs in $\Omega\left(\mathbf{t}^u, k\right)$ actually appear in the user's next k tracks:

$$\begin{aligned} &\textit{precision@}k = \\ &\frac{1}{|U|} \sum_{u \in U} \frac{\left| \left\{ s \colon s \in \Omega\left(\mathbf{t}^u, k\right) \wedge s \in \left(s^u_{101:100+k}\right) \right\} \right|}{k}. \end{aligned} \tag{6.18}$$

We can also generalize this to *precision@*$[j:k]$:

$$\begin{aligned} &\textit{precision@}[j:k] = \\ &\frac{1}{|U|} \sum_{u \in U} \frac{\left| \left\{ s \colon s \in \Omega\left(\mathbf{t}^u, k-j+1\right) \wedge s \in \left(s^u_{100+j:100+k}\right) \right\} \right|}{k-j+1}. \end{aligned} \tag{6.19}$$

Here we disregard the user's first next $j-1$ tracks, since it is often easier to predict the immediate next tracks than it is to predict tracks further in the future. For the next results, we also slightly alter the definition of $\Omega\left(\mathbf{t}^u, k\right)$: given the fact that no song in $\left(s^u_{101:150}\right)$ occurs in $\left(s^u_{1:100}\right)$ for all users u, as described in Section 6.4.2, we only regard the k nearest songs of $\mathbf{t}^u$ that do not appear in $\left(s^u_{1:100}\right)$. For the classification models, we simply take the top k songs with the highest scores, and compare them to the ground truth. If we denote these top k songs by $\Omega\left(\mathcal{R}\left(\mathbf{s}^u_{1:k}; \mathbf{W}\right), k\right)$, we can reuse the same definition of *precision@k* from above.

The results of the experiments are shown in Table 6.5. In bold we mark the best performing model for each task, and the second best model is underlined. The overall precisions are quite low, but given that we aggressively filtered the listening data (Section 6.4.2), the task is rather difficult. The history-based RNNs clearly perform best in all tasks. Generally, for *precision*@10, @25 and @50 all short-term models outperform the long-term models. But once we skip the first 25 songs, which are easier to predict, the long-term models take over, which shows that listening behavior indeed changes over time. The performance of the playlist RNNs and weight-based models are comparable to the exponential discount models, which we already saw in Figure 6.5.

All four classification models that we trained achieved the same precision score of 0 percent, so we listed them as one entry in Table 6.5. They

Table 6.5: Results for the **precision@k** experiments on filtered listening history data.

	Precision (%)				
	@10	@25	@50	@[25 : 50]	@[30 : 50]
rPST	1.64	2.39	2.81	1.15	0.94
rPLT	1.27	1.98	2.64	1.30	1.06
rHST	**2.03**	**2.95**	**3.72**	1.85	1.53
rHLT	1.40	2.31	3.25	**1.89**	**1.63**
bWST	1.67	2.37	2.94	1.28	1.04
bWLT	1.32	1.95	2.62	1.31	1.06
$\gamma = 0.85$	1.94	2.63	3.00	1.20	0.96
$\gamma = 0.97$	1.56	2.20	2.77	1.30	1.05
$\gamma = 1.0$	1.16	1.78	2.41	1.24	1.02
Classification	0.00	0.00	0.00	0.00	0.00

were not able to correctly guess any of the 50 future songs a user might listen to. We can think of many reasons why we see this result. First, the output dimensionality of 6 million is extremely large, which makes it difficult to discriminate between different but comparable items. The RecSys Challenge 2015 dataset, used in both the works of Hidasi et al. and Tan et al., only has around 37,500 items to predict, which is 160 times less than 6 million. Second, the number of weights in the classification model is orders of magnitudes larger than for the regression model, which causes learning to be much harder. And third, the data in the works by Hidasi et al. and Tan et al. comes from user sessions, which are mostly contained and coherent sequences of songs a user listens to within a certain time span, see also Section 6.2. The listening history dataset in our work goes across user sessions to be able to recommend on the long term, which makes it much more difficult to model the temporal dynamics. This is reflected in the overall low precision accuracies.

A note on scalability

As indicated in the title of this article, our methodology should allow for scalable music discovery. The training procedure—i.e. training word2vec and the RNN—is not time-critical and can be trained offline. Despite the fact that these procedures could be parallelized when needed [37], we will focus on the recommendation part of the system itself, which is more time-critical.

Since every user can be treated independently, the entire pipeline we have proposed in Figure 6.1 is 'embarrassingly parallel' and can therefore

be scaled up to as many computational nodes as available. Retrieving song vectors comes down to a dictionary lookup in constant time $O(1)$. Calculating user taste vectors through the RNN is linear $O(n)$ in the number of historical song vectors n we consider. An extensive study of the scalability of Annoy, the last part in the pipeline, is beyond the scope of this paper, and poses a trade-off between accuracy and performance: more tree nodes inspected leads generally to more accurate nearest neighbors, but a slower retrieval time (approximately linear in the number of inspected nodes)[3]. Retrieving 1,000 nearest neighbors using 10 trees and 10,000 inspected nodes only takes on average 2.6ms on our system, which is in same order of magnitude compared to the RNN prediction times given in Table 6.2.

Combining all the above, sampling a taste vector from the rHLT RNN and retrieving the top 50 closest songs from the Annoy LSH tree over 1,000 runs takes on average 58ms on our system, while retrieving the top 50 songs from the BPR RNN takes on average 754ms, which is 13 times slower.

6.6 Conclusions

We modeled users on large-scale online music streaming platforms for the purpose of new music discovery. We sequentially processed a user's listening history using recurrent neural networks in order to predict a song he or she will listen to in the future. For this we treated the problem as a regression rather than classification task, in which we predict continuous-valued vectors instead of distinct classes. We designed a short-term and long-term prediction model, and we trained both versions on playlist data as well as filtered user listening history data. The best performing models were chosen to be as small and efficient as possible in order to fit in large-scale production environments. Incorporating extra play context features did not significantly improve the models. We performed a set of experimental analyses for which we conclude that the history-based models outperform the playlist-based and all baseline models, and we especially pointed out the advantages of using the regression approach over the classification baseline models. We also saw that there is indeed a difference between short-term and long-term listening behavior. In this work we modeled these with different models. One possible line of future work would be to design a single sequence-to-sequence model that captures both short and long term time dependencies to predict the entire future listening sequence [5].

[3]There is an excellent web article by Radim Rehurek from 2014 which studies this in depth, see rare-technologies.com/performance-shootout-of-nearest-neighbours-querying

References

[1] F. Figueiredo, B. Ribeiro, C. Faloutsos, N. Andrade, and J. M. Almeida. *Mining Online Music Listening Trajectories.* In ISMIR, 2016.

[2] O. Barkan and N. Koenigstein. *Item2vec - Neural Item Embedding for Collaborative Filtering.* RecSys Posters, 2016.

[3] D. Liang, J. Altosaar, and L. Charlin. *Factorization Meets the Item Embedding: Regularizing Matrix Factorization with Item Co-occurrence*. In ICML Workshop, 2016.

[4] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. *Distributed Representations of Words and Phrases and their Compositionality*. In NIPS 2013: Advances in neural information processing systems, October 2013.

[5] I. Sutskever, O. Vinyals, and Q. V. Le. *Sequence to Sequence Learning with Neural Networks*. In NIPS 2014, September 2014.

[6] D. Bahdanau, K. Cho, and Y. Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. In ICLR, 2015.

[7] A. Graves and N. Jaitly. *Towards End-To-End Speech Recognition with Recurrent Neural Networks.* In ICML, pages 1764–1772, 2014.

[8] J. Bennett and S. Lanning. *The Netflix Prize*. In Proceedings of KDD cup and workshop, 2007.

[9] Y. Koren, R. Bell, and C. Volinsky. *Matrix Factorization Techniques for Recommender Systems*. Computer, 42(8):30–37, 2009.

[10] G. Dror, N. Koenigstein, and Y. Koren. *Yahoo! music recommendations - modeling music ratings with temporal dynamics and item taxonomy.* In RecSys, pages 165–172, New York, New York, USA, 2011. ACM.

[11] Y. Hu, Y. Koren, and C. Volinsky. *Collaborative filtering for implicit feedback datasets*. IEEE International Conference on Data Mining, pages 263–272, 2008.

[12] A. Paterek. *Improving regularized singular value decomposition for collaborative filtering*. In Proceedings of KDD cup and workshop, 2007.

[13] D. D. Lee and H. S. Seung. *Algorithms for Non-negative Matrix Factorization.* NIPS 2000, 2000.

[14] R. Pan, Y. Zhou, B. Cao, N. N. Liu, R. Lukose, M. Scholz, and Q. Yang. *One-Class Collaborative Filtering*. In 2008 Eighth IEEE International Conference on Data Mining, pages 502–511. IEEE, 2008.

[15] C. C. Johnson. *Logistic matrix factorization for implicit feedback data*. In NIPS 2014 Workshop on Distributed Machine Learning . . . , 2014.

[16] J. L. Moore, S. Chen, D. Turnbull, and T. Joachims. *Taste Over Time - The Temporal Dynamics of User Preferences.* In ISMIR, 2013.

[17] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. *WaveNet: A Generative Model for Raw Audio*. arXiv.org, September 2016. arXiv:1609.03499v2.

[18] T. Sercu and V. Goel. *Advances in Very Deep Convolutional Neural Networks for LVCSR.* In Interspeech, 2016.

[19] A. Karpathy, J. Johnson, and L. Fei-Fei. *Visualizing and Understanding Recurrent Networks*. arXiv.org, June 2015. arXiv:1506.02078v2.

[20] A. L. Maas, A. Y. Hannun, and A. Y. Ng. *Rectifier nonlinearities improve neural network acoustic models*. In ICML, 2013.

[21] S. Hochreiter and J. Schmidhuber. *Long short-term memory*. Neural Computation, 9(8):1735–1780, 1997.

[22] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. *LSTM: A Search Space Odyssey*. arXiv.org, March 2015. arXiv:1503.04069v1.

[23] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.* arXiv.org, 2014. arXiv:1412.3555v1.

[24] B. Hidasi, A. Karatzoglou, L. Baltrunas, and D. Tikk. *Session-based Recommendations with Recurrent Neural Networks.* arXiv.org, 2016. arXiv:1511.06939v4.

[25] Y. K. Tan, X. Xu, and Y. Liu. *Improved Recurrent Neural Networks for Session-based Recommendations.* arXiv.org, 2016. arXiv:1606.08117v2.

[26] M. G. Ozsoy. *From Word Embeddings to Item Recommendation*. arXiv.org, January 2016. arXiv:1601.01356v3.

[27] A. Van Den Oord, S. Dieleman, and B. Schrauwen. *Deep content-based music recommendation*. In NIPS, pages 2643–2651, 2013.

[28] F. Hill, K. Cho, A. Korhonen, and Y. Bengio. *Learning to Understand Phrases by Embedding the Dictionary.* TACL, 2016.

[29] D. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. In ICLR, 2015.

[30] M. Charikar. *Similarity estimation techniques from rounding algorithms.* In STOC, page 380, New York, New York, USA, 2002. ACM Press.

[31] R. Al-Rfou, G. Alain, A. Almahairi, and e. al. *Theano - A Python framework for fast computation of mathematical expressions.* arXiv.org, 2016. arXiv:1605.02688v1.

[32] C. N. dos Santos and M. Gatti. *Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts*. In COLING 2014, the 25th International Conference on Computational Linguistics, pages 69–78, Dublin, July 2014.

[33] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. *Natural Language Processing (Almost) from Scratch*. The Journal of Machine Learning Research, 12, February 2011.

[34] C. De Boom, S. Van Canneyt, T. Demeester, and B. Dhoedt. *Representation learning for very short texts using weighted word embedding aggregation*. Pattern Recognition Letters, 80(C):150–156, September 2016.

[35] J. Goodman. *Classes for fast maximum entropy training*. In 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221, pages 561–564 vol.1. IEEE, 2001.

[36] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme. *BPR - Bayesian Personalized Ranking from Implicit Feedback.* UAI, 2009.

[37] S. Ji, N. Satish, S. Li, and P. Dubey. *Parallelizing Word2Vec in Multi-Core and Many-Core Architectures.* CoRR, cs.DC, 2016.

7

Conclusions and Future Research Directions

"Do it for your people, do it for your pride. How you ever gonna know if you never even try? Do it for your country, do it for your name. 'Cause there gonna be a day when you're standing in the hall of fame."

—The Script, 2012

In the introduction of this book I have talked extensively about why we, as human beings, are able to interpret the environment around us and how we can interact with it. Our brain holds many complex world models that process the information we get from our senses, and we are constantly using these models to test hypotheses about the future through the mechanisms of anticipation and expectation. We are still not able to fully explain and understand how all this information, knowledge and reasoning is digested in the human brain, and this will remain an important research topic for the years to come.

Artificial intelligence and machine learning are modern research fields that have the strongest ties with computer science and mathematics, but also borrow insights from biology, psychology, philosophy, etc. Through extensive research, many of the existing machine learning models are now approaching or even surpassing human-level behavior on a couple of tasks such as object recognition and playing games. In this doctoral thesis, I have focused on those machine learning models that are able to process and

learn from sequential data, such as text, music, playlists, Twitter streams, etc. similar to how humans would analyze this data. Interpreting and writing text is usually an easy task for humans, because we have been trained in this particular field for as long as we can remember. On the other hand, for many of us, analyzing and composing music is a rather difficult assignment, because we have little knowledge about musical patterns and music theory, which is often only reserved for experienced musicians and composers. Recommending songs, videos, clothes, etc. is not straightforward either, because it requires extensive domain knowledge. A lot of us can probably recall a particular retail store we often go to because of the useful and (hopefully honest) advice the seller is providing to us. Even then it is impossible to serve thousands or even millions of customers every day while maintaining the same high-quality service level. And analyzing the entire Twitter stream to look for event-related tweets is a task that is nearly impossible for humans to execute. This is because of the amount of data that is produced every second, which is just too high to process manually. We have then entered the realm of so-called *big data analytics*, which requires, next to the machine learning research itself, also a lot of engineering effort.

It is needless to say that this doctoral thesis has only just scratched the surface of the above-mentioned applications. In the coming years the area of research around machine learning, deep learning and, in particular, sequential modeling will continue to attract a multitude of researchers. According to recent predictions, the field will keep growing extensively and will gain importance throughout many industrial branches. Being able to extract high-quality knowledge from data and to use it intelligently and wisely for the purpose of marketing, retail, process optimization, business insights, etc. will create a huge boost for anyone's company. I will now briefly go over the main conclusions of this book, after which I will list a set of possible future research tracks and open challenges. In particular, I will focus on two domains: text to knowledge, and recurrent hierarchies.

Text to knowledge With the introduction of word2vec we are now able to relate words to each other on a semantic level. In Chapter 3 we have designed an out-of-the-box algorithm to combine word embeddings into one single sentence embedding. We have shown that our method was able to outperform naive sentence embedding schemes, as well as the tf-idf representation that was used in Chapter 2. A sentence embedding that is based on word2vec helps us to identify different texts that share a certain semantic context, such as movie reviews or weather forecasts. If we were

to include more dimensions in the word2vec space, we would be able to distinguish more detailed and fine-grained contexts from each other, such as the difference between reviews for scifi, comedy, drama... movies.

One of the flaws of plain word2vec, however, is that there is no detection or clear distinction made between words that appear in different contexts, such as 'apple'—the fruit, the tech company, the 'Big Apple', etc. Another important observation is that relationships between words are practically lost, such as word hierarchies, synonyms, fixed expressions, etc. For example, with word2vec we can discover that the words 'father', 'mother', 'son', 'daughter', 'siblings', 'uncle', 'grandfather'... are semantically related to each other, but we are unable to construct a detailed family tree. The same goes for 'Obama', 'Clinton' and 'Bush': all three men were presidents of the United States, but we don't know the order in which they held office, which political party each of them belonged to, etc. How exactly we can extract these relationships from text and represent them in the embedding space, or how we can learn to distinguish between words with multiple meanings, are important research questions for the future—although various research labs have been or are already looking into these challenges.

Recurrent hierarchies In Chapter 4, the training and sampling principles behind recurrent neural networks (RNNs) were discussed in detail. In the subsequent chapters, we have used RNNs for the purpose of piano music composition and song recommendations. By learning from a large collection of music, we have shown that these types of deep networks are able to capture and model melodic fragments as well as harmonic progressions. And on the Spotify platform we were able to capture the listening behavior of users by processing the sequence of songs they have listened to. As a result of this, low-dimensional user embeddings were created that can be used to recommend new songs in an efficient and scalable manner.

A particular issue that we have not addressed in this thesis, is that many of the real-life sequences we encounter follow some kind of hierarchical structure. For example, if we look at any modern pop song, we notice that it is composed of an intro, verses, repeated choruses, and potentially a bridge and outro. The parts themselves are also built up of series of bars, mostly multiples of 4 or 8, accompanied by some fixed chord progressions. Each of these bars contains melodic fragments, also called 'themes', that can be short or spread out across multiple bars. These themes are repeated throughout the entire piece, in one form or another. What we therefore see is that a pop song is a hierarchical structure of themes, chord progressions, and choruses and verses. If, for a moment, we take a look at users on

Spotify, we also notice a hierarchy in their listening behavior as well. At the lowest level we have the separate tracks. One level up, these tracks appear in separate albums, playlists or have been released by the same artist. And these albums might all belong to the same genre, or each to a different genre. The listening behavior of a user is therefore a walk through genres, within these genres through albums and artists, and on the most fine-grained level it is a walk through tracks.

As we have discussed in the introductory chapter, deep neural networks that process images are able to model features in a hierarchical fashion, e.g. lines, shapes, eyes, faces, ... That is, after all, what we observe from the learned filters in these networks. For recurrent neural networks, on the other hand, it is less apparent whether stacked layers are able to extract a sound hierarchical structure from the presented sequences. And although LSTMs and GRUs are designed to counter the problem of vanishing gradients, thereby enabling us to process longer sequences, in practice it turns out that, at least in the context of music, important information tends to get lost anyway after a few hundred or more tokens[1]. In future research on RNNs, and on sequence modeling in general, it will therefore be important to look further into sequence hierarchy. Either by explicitly modeling the hierarchy with the help of specific domain knowledge, or by creating novel deep recurrent architectures that are stimulated to look for hierarchies in sequences. Next to this, it will also become more important to be able to retain specific pieces of information across more than hundreds or thousands of tokens in the sequence, such as melodic themes in a musical composition. For this purpose, I mainly look towards RNNs with specific memory cells, so-called *memory networks*. Although there has been done extensive research in this field, I believe that their power still has to be proven in the context of music generation.

As a final personal remark, I believe that the power of RNNs has not yet been explored or exploited to its full extent. They will continue to amaze us in the context of various applications and research domains. And once we are able to figure out how to deploy them efficiently on low-powered end devices such as smartphones, watches, sensors, etc. this will further open up the already wide range of deep learning applications.

[1]Claim based on a conversation with Aiva Technologies, a Luxemburg-based start-up company that composes movie and game soundtracks.