Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Žiga Sajovic

Operatorski račun nad programskimi prostori

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Borut Robič

Ljubljana, 2017

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Razvijte algebraični jezik, ki bo deloval kot formalni račun za globoko učenje, hkrati pa bo orodje za proučevanje programov, ki so v njem implementirani. Jezik naj deluje kot poln model globokega učenja. Omogoča naj tako izražanje programov, da bo že njihov zapis nudil teoretični vpogled vanje.

Avtor se zahvaljuje Mariu Medvedu iz podjetja XLAB za vero in podporo pri razvoju ubesedene teorije, Martinu Vuku za potrpežljivost ob izraznih izomorfizmih, Juretu Kališniku za preverbo dokazov ter Borutu Robiču za vodstvo skozi akademsko življenje.

Za Barbi.

Kazalo

Povzetek

Naslov: Operatorski račun nad programskimi prostori

Avtor: Žiga Sajovic

V delu razvijemo algebraični jezik, ki predstavlja formalni račun za globoko učenje, in je hkrati model, v katerem je programe mogoče tako implementirati kot tudi preučevati.

V ta namen razvijemo abstraktni *računski model avtomatsko odvedljivih programov*. V njem so programi elementi t. i. *programskih prostorov*. Programe gledamo kot preslikave končno-dimenzionalnega vektorskega prostora vase, imenovanega *navidezni pomnilniški prostor*. Navidezni pomnilniški prostor je algebra programov, torej *algebraična podatkovna struktura* (s katero je mogoče računati). Elementi navideznega pomnilniškega prostora pa omogočajo razvoj programov v neskončne tenzorske vrste. Na programskih prostorih definiramo *operator odvajanja*, s pomočjo njegovih potenc pa implementiramo *posplošen operator premika* in *operator kompozicije* programov.

Tako konstruiran algebraični jezik je poln model globokega učenja. Omogoča takšen način izražanja programov, da že njihov zapis nudi teoretični vpogled vanje.

Ključne besede: operatorska algebra, tenzorska algebra, nevronske mreže, globoko učenje, avtomatsko odvajanje.

Abstract

Title: Operational Calculus on Programming Spaces

Author: Žiga Sajovic

In this work we develop an algebraic language that represents a formal calculus for deep learning and is, at the same time a model which enables implementations and investigations of programs.

To this purpose, we develop an *abstract computational model of automatically differentiable programs.* In the model, programs are elements of op. cit. *programming spaces.* Programs are viewed as maps from a finite-dimensional vector space to itself op. cit. *virtual memory space.* Virtual memory space is an algebra of programs, *an algebraic data structure* (one can calculate with). The elements of the virtual memory space give the expansion of a program into an infinite tensor series. We define a *differential operator* on programming spaces and, using its powers, implement the *general shift operator* and the *operator of program composition.*

The algebraic language constructed in this way is a complete model of deep learning. It enables us to express programs in such a way, that their properties may be derived from their source codes.

Keywords: operator algebra, tensor algebra, neural networks, deep learning, automatic differentiation.

1

Uvod

Zaradi diskretne narave računalniške znanosti prihaja do vrzeli med matematično analizo in programiranjem. Medtem ko se je algebra že vpletla v veje programiranja preko algebraičnih podatkovnih tipov funkcijskih jezikov, njena uporaba ostaja predvsem v problemih, ki so diskretni in kombinatorični. A zaradi hitrosti razvoja strojnega in globokega učenja se pojavlja potreba po modelih programiranja, ki bi omogočali obravnavo (odsekoma) zveznih in odvedljivih programskih struktur [8]. Tej potrebi delno že zadoščajo sodobne metode avtomatskega odvajanja. Te metode omogočajo učinkovit izračun odvodov matematičnih funkcij, implementiranih v obliki računalniških programov [26]. Vendar pa ne omogočajo algebraičnega manipuliranja s programi in tako tudi ne vplivajo na bistvo polemike o razkolu (ki je v nezmožnosti algebraičnega manipuliranja odvedljivih programov).

Namen tega dela je zapolniti vrzel med programiranjem in analizo, ki je značilna za trenutno stanje [25].

V drugem poglavju (*Motivacija in ozadje*) bomo predstavili, kako lahko *delno* prevedemo *algebraične podatkovne tipe* funkcijskih jezikov (kot npr. Haskell) v vsote in produkte elementov neke algebraične strukture. Razgalili bomo probleme v teh jezikih, ki so razlog, da je prevajanje le *delno*. Izpostavili bomo tudi slabost *globokega učenja*, kjer zaradi pomanjkanja formalnega aparata obravnava poteka predvsem empirično in s tem utemeljili potrebo po rezultatih svojega dela. Opisali bomo tudi pomen svojega dela za področje *avtomatskega odvajanja* programov, ki je v splošni uporabi v uporabnem računalništvu (npr. v inženiringu, simulacijah).

V tretjem poglavju (*Programski prostori*) pričnemo z vzpostavitvijo potrebnih objektov za konstrukcijo takšnega algebraičnega jezika. Programe interpretiramo kot preslikave vektorskih prostorov vase in na podlagi takšne interpretacije definiramo *avtomatsko odvedljive programe*.

Svoj računski model razširimo v četrtem poglavju (*Odvedljivi programski prostori*), ko konstruiramo *navidezni pomnilniški prostor*. Slednji deluje kot *algebraična podatkovna struktura*, tj. podatkovna struktura, s katero se da računati. Tako podatkovno strukturo potrebujemo za konstrukcijo t. i. *odvedljivih programskih prostorov*, ki jih strogo definiramo in dokažemo potrebne izreke o njihovi zaprtosti. Poglavje zaključimo z definicijo *navideznih tenzorskih strojev*, ki konstruirajo odvedljive programske prostore, in z *implementacijo* odvedljive različice jezika *C*++, ki je dostopna na avtorjevi strani [32].

V petem poglavju (*Operatorski račun nad programskimi prostori*) na podlagi prej vpeljanih programskih prostorov razvijemo algebraični jezik, ki deluje kot formalni račun globokega učenja in je hkrati model, v katerem je programe mogoče tako implementirati kot tudi preučevati. Nato izpeljemo splošni *razvoj programov v tenzorsko vrsto* ter izpeljemo še operatorja *premika* in *kompozicije* programov. Slednja omogočata konstrukcijo iteratorjev in komponerjev. Pokažemo, kako vsi ti operatorji delujejo kot abstrakcije, ki omogočajo, da se izračuni opravijo samo z operatorji, preden te apliciramo na konkretni program – s tem se izognemo delu s tenzorskimi vrstami. Nato razvijemo preslikavo redukcije reda in s tem rešimo problem *avtomatsko odvedljivih programov* [25], saj omogoča implementacije programov, ki operirajo na odvedljivih odvodih drugih programov. Z vpeljanimi operatorji pokažemo, da naša teorija uvaja poln (samozadosten) model globokega učenja. Ta model omogoča tak način izražanja programov, da že njihov zapis razgalja njihove lastnosti. Poglavje zaključimo z obravnavo pomena svoje teorije za kontrolne strukture in z napotki za uporabo teorije v praksi.

Diplomsko delo temelji na avtorjevih člankih [35, 34] in delu [32, 33], kamor bralca usmerimo v zaključnem poglavju (*Sklep in nadaljnje delo*), v katerem navedemo možnosti, ki jih izpeljana teorija še ponuja.

2

Motivacija in ozadje

"Von Neumann languages do not have useful properties for reasoning about programs. Axiomatic and denotational semantics are precise tools for describing and understanding conventional programs, but they only talk about them and cannot alter their ungainly properties. Unlike von Neumann languages, the language of ordinary algebra is suitable both for stating its laws and for transforming an equation into its solution, all within the language."

– John Backus, *Can Programming Be Liberated From the von Neumann Style?*

Po prejetju Turingove nagrade leta 1977 za delo na prevajalniku za programski jezik Fortran je imel John Backus svoje znamenito predavanje, kot nekakšno opravičilo za svoje delo. Namreč, v predavanju je kritiziral von Neumannove jezike (katerih predstavnik je ravno Fortran), in tudi von Neumannovo računalniško arhitekturo. Toda segel je še globlje in se obregnil ob neobstoj nekakšne algebre programov, ki bi programskim jezikom omogočala, da bi sklepali o programih, ki jih implementirajo [3]. V prihodnjih letih je računalniška znanost pričela z raziskavami Backusovih skrbi. Njegovo skrb glede zaporednega izvajanja, tj. von Neumannovega ozkega grla, dodobra naslavljajo nedavni napredki na področju strojne opreme in arhitekture, ki danes streže tudi globokemu učenju (npr. Nvidia, Google TPU). Vendar njegova skrb algebraičnega obravnavanja programov še vedno tli, čeprav jeziki, kot je Haskell, s pomočjo teorije kategorij implementirajo algebraične podatkovne tipe. Povezavo takih podatkovnih tipov s klasično algebro in njihove pomanjkljivosti bomo spoznali v razdelku 2.1.

Globoko učenje je programska paradigma, ki zaradi svoje algebraične naravnanosti ponuja možnost formalizma, ki naslavlja obe Backusovi skrbi. Trenutno je globoko učenje še vedno v pomanjkanju formalizma, zato večinoma temelji na empiričnih spoznanjih. Področje bomo na kratko spoznali v razdelku 2.2 in izpostavili to pomanjkanje formalizma. Razvoj algebraičnega jezika, ki vsebuje globoko učenje, je plod avtorjevega dela zadnjih let [35] [34]. Ker je področje prežeto z gradienti in Jacobiani, je področje soočeno s potrebo po jeziku, ki bi omogočal sklepanje o analitičnih lastnostih programov zgolj z uporabo algebraičnih prijemov. Kot bomo videli, tej potrebi lahko zadostimo z uporabo tenzorske algebre in operatorskega računa, predstavljenega v našem delu. Tak formalizem pa je tudi orodje, ki presega globoko učenje in teoretična preučevanja programskih jezikov, saj vsebuje posplošitev in poenotenje metod avtomatskega odvajanja (razdelek 2.4).

2.1 Algebra algebraičnih podatkovnih tipov

Predmet tega razdelka je algebra algebraičnih podatkovnih tipov in njene lastnosti ter pomanjkljivosti, ki jih bomo za lažje razumevanje prikazali na primeru Haskella. Slednje je povzeto po zapisih [30].

V splošnem je algebrska struktura množica objektov in operacij nad njimi, ki omogočajo transformacije teh objektov v nove objekte. V algebri tipov jezika Haskell so objekti podatkovni tipi, denimo *Bool* in *Int*, operacije pa gradijo iz danih tipov nove, kompleksnejše tipe. Primer je konstruktor *Either*, ki izgradi nov tip *Either Bool Int* iz tipov Bool in Int.

2.1.1 Operacije

Preštevanje

Povezavo z algebro števil lahko vzpostavimo s preštevanjem možnih vrednosti, ki jih tip lahko zasede. Na primeru *Bool*, definiranim z

$$data\ Bool = False \mid True,$$

to pomeni, da lahko tip *Bool* zasede dve vrednosti (lahko bi zasedel tudi *undefined*, a tej vrednosti se v prid razlagi izognemo). Tako prispemo do nove definicije.

Definicija 2.1 (Izomorfnost tipov) *Izomorfnost (===) dveh tipov* $Type_1$ *in* $Type_2$ *definiramo s*

$$Type_1 === Type_2 \iff |Type_1| = |Type_2|,$$

kjer $|Type|$ *označuje število vrednosti, ki jih tip lahko zaseda.*

Če sta dva tipa izomorfna po definiciji 2.1, med njima obstaja bijektivna preslikava. Na zgornjem primeru to pomeni

$$Bool === 2,$$

kar bomo podrobneje razjasnili, ko bomo spoznali aditivne tipe.

Če je *Bool* enak 2, čemu je enaka 1? V literaturi so takšni objekti navadno poimenovani z *Unit*; v Haskellu je Unit označen z *()*.

Vsota

Če sta a in b tipa, potem je tip *Add* a b, ki ustreza njuni vsoti, definiran z

$$data\ Add\ a\ b = AddL\ a \mid AddR\ b.$$

Torej, tip *Add* a b je oblika unije[1], ki vsebuje bodisi a ali b. Zakaj tip *Add* a b ustreza vsoti $a + b$ tipov a in b je razvidno iz preštevanja. Na primer, tip

[1]ang. tagged union

Add Bool () vsebuje $|Bool| + |()| = 2 + 1 = 3$ vrednosti. V Haskellu je tipu *Add a b* izomorfen tip *Either a b*, a v našem delu je primernejše poimenovanje *Add*.

Produkt

Če sta a in b tipa, potem je tip $Mul\ a\ b$, ki ustreza njunemu produktu, definiran z

$$data\ Mul\ a\ b = Mul\ a\ b.$$

Torej, tip $Mul\ a\ b$ je zbirnik, ki drži tako a kot b. Zakaj tip $Mul\ a\ b$ ustreza produktu $a \cdot b$ tipov a in b, je razvidno iz preštevanja. Na primer, Mul *Bool Bool* vsebuje $|Bool| \cdot |Bool| = 2 \cdot 2 = 4$ vrednosti. V Haskellu je tipu $Mul\ a\ b$ izomorfen tip *(a.b)*, a v našem delu je primernejše poimenovanje *Mul*.

Ničla – identiteta za operacijo vsote

Z vsoto in produktom lahko iz tipa *Unit* generiramo tipe, ki ustrezajo pozitivnim celim številom. Še vedno pa manjka tip, ki bi ustrezal številu 0, torej tip brez vrednosti. Tak tip definiramo z

$$data\ Void.$$

Ker definicija ne vsebuje konstruktorja, vrednosti tipa *Void* ni mogoče konstruirati – to pomeni, da ima *nič* vrednosti.

2.1.2 Zakoni v algebri tipov

Kakor v algebri števil, tudi v algebri tipov zakoni določajo enakost dveh objektov (po definiciji enakosti 2.1). V korist jasnosti navajamo primer implementacije izomorfizma

$$Bool === Add\ ()\ (),$$

ki jo vidimo na sliki 2.1.2.

```haskell
to :: Bool -> Add () ()
to False = AddL ()
to True  = AddR ()

from :: Add () () -> Bool
from (AddL _) = False
from (AddR _) = True
```

Slika 2.1: Primer izomorfizma

Zakoni vsote tipov

Navajamo primer dveh zakonov vsote tipov.
Prvi veli

$$Add\ Void\ a === a,$$

da ima tip *Add Void a* toliko vrednosti kot tip *a*.
Drugi veli

$$Add\ a\ b === Add\ b\ a,$$

da je vrstni red operandov nepomemben.
Oba zakona sta bralcu verjetno bolj poznana iz algebre števil, kjer se glasita

$$0 + a = a,$$

in

$$a + b = b + a.$$

Zakona za algebro tipov bi lahko dokazali s konstrukcijo, kot je bilo storjeno na sliki 2.1.2, a to prepuščamo bralcu.

Zakoni produkta tipov

Navajamo tri koristne zakone, ki se nanašajo na produkt tipov.
Prvi veli

$$Mul\ Void\ a === Void,$$

da je produkt poljubnega tipa s tipom *Void* enak tipu *Void*, kar je moč interpretirati na sledeč način: ker je nemogoče konstruirati objekt tipa *Void*, je nemogoče konstruirati urejen par tipov, ki bi vseboval tip *Void*.
Drugi zakon veli

$$Mul\ ()\ a === a,$$

da je toliko urejenih parov, ki imajo tip *Unit* na prvem mestu, kot je vrednosti tipa a, ki je na drugem mestu.
Tretji zakon veli

$$Mul\ a\ b === Mul\ b\ a,$$

da je število urejenih parov dveh tipov enako, ne glede na vrstni red tipov. Ti zakoni so bralcu verjetno bolj poznani iz algebre števil, kjer se glasijo

$$0 \cdot a = 0, \tag{2.1}$$

$$1 \cdot a = a, \tag{2.2}$$

$$a \cdot b = b \cdot a. \tag{2.3}$$

Iz zgornjih zakonov algebre tipov je možno izpeljati tudi druge zakone, ki jih poznamo iz algebre števil, denimo zakon distributivnosti

$$Mul\ a\ (Add\ b\ c) === Add\ (Mul\ a\ b)\ (Mul\ a\ c),$$

oziroma

$$a \cdot (b + c) = a \cdot b + a \cdot c.$$

2.1.3 Funkcijski tipi

Ob konkretnih tipih, kot sta *Int* in *Bool*, poznamo tudi funkcijske tipe, denimo *Int* $\to$ *Bool*. V algebro tipov jih umestimo preko preštevanja. Koliko funkcij tipa $a \to b$ obstaja? Za pojasnitev navajamo primer. Vzemimo tip s tremi vrednostmi,

$$data\ Trio = prvi \mid drugi \mid tretji,$$

in preštejmo vse funkcije tipa

$$Trio \to Bool.$$

Po kratkem razmisleku je očitno, da je teh funkcij $|Bool|^{|Trio|} = 2^3 = 8$, oziroma v splošnem

$$|B^A| = |B|^{|A|},$$

kjer B^A označuje množico vseh funkcij tipa $A \to B$.

Zakoni funkcijskih tipov

Obstajata dva zakona funkcijskih tipov, ki vključujeta enoto (*Unit*).
Prvi veli

$$() \to a === a,$$

da je toliko funkcij tipa *()*$\to a$, kot je vrednosti tipa a.
Drugi veli

$$a \to () === (),$$

da obstaja natanko ena funkcija tipa $a \to$ *()*; označimo jo s *const ()*.
V algebri števil sta to zakona

$$a^1 = a,$$

in

$$1^a = 1.$$

Obstaja pa tudi zakon, ki dovoljuje faktorizacijo skupnih argumentov. Označimo z $(a \to b, a \to c)$ urejen par dveh funkcijskih tipov $a \to b$ in $a \to c$. Potem zakon pravi

$$(a \to b, a \to c) === a \to (b, c),$$

kjer $a \to (b, c)$ označuje funkcijski tip, ki slika iz a v urejen par (b, c). V algebri števil je ustrezni zakon

$$b^a \cdot c^a = (b \cdot c)^a.$$

Prav tako obstaja zakon o funkcijah, ki vračajo funkcije:

$$a \to (b \to c) === (b, a) \to c,$$

katerega oblika v algebri števil je

$$(c^b)^a = c^{b \cdot a}.$$

2.1.4 Rekurzivni tipi

V prejšnjem razdelku smo vpeljali operacije nad tipi ter navedli njihove osnovne lastnosti. S tem smo vzpostavili algebro tipov. V tem razdelku pokažemo dvoje: najprej uporabo algebre tipov nad kompleksnejšimi tipi, potem pa pomanjkljivosti, ki jih zaznamo v tej algebri.

Seznami

Ena osnovnejših struktur večine programskih jezikov je *seznam*. Seznam $L\ a$ objektov tipa a je bodisi prazen (*None*) ali pa je sestavljen iz objekta tipa a, ki je pripet na obstoječ seznam tipa a. To rekurzivno (induktivno) definicijo na kratko izrazimo z

$$data\ L\ a = None \mid a\ (L\ a). \tag{2.4}$$

V algebri tipov, vpeljani v prejšnjih razdelkih, seznam izrazimo kot

$$L(a) = 1 + a \cdot L(a).$$

Enačbo lahko pričnemo razvijati, kjer bomo izraze tipa $a \cdot a$ nadomestili z a^2:

$$\begin{aligned}
L(a) &= 1 + a \cdot L(a) \\
L(a) &= 1 + a \cdot (1 + a \cdot L(a)) \\
L(a) &= 1 + a + a^2 \cdot (1 + a \cdot L(a)) \\
L(a) &= 1 + a + a^2 + a^3 + \cdots . \qquad (2.5)
\end{aligned}$$

Izraz (2.5) pravi, da obstaja natanko en prazen seznam (člen 1), natanko en seznam z enim elementom (člen a), natanko en seznam z dvema elementoma (člen a^2), itd.

Poznavanje rodovnih funkcij in formalnih potenčnih vrst nam (skupaj z zakoni prejšnjega razdelka) omogoča tudi drug način kodiranja informacije, zapisane v izrazu (2.4):

$$L(a) = 1 + a \cdot L(a)$$

$$\Longrightarrow$$

$$L(a) = \frac{1}{1-a} \tag{2.6}$$

$$\Longrightarrow$$

$$L(a) = \sum_{i=0}^{\infty} a^i,$$

kar je identično dognanje kot rekurzivni razvoj v (2.5). Tukaj poudarimo, da so algebraične manipulacije v (2.6) dovoljene, saj gre le za drug način kodiranja iste informacije. Pri splošnih algebraičnih manipulacijah tipov, kjer z manipuliranjem enakosti izražamo nove enakosti (ne le druge oblike začetnih enakosti), lahko naletimo na težave; to bomo spoznali v razdelku 2.1.5.

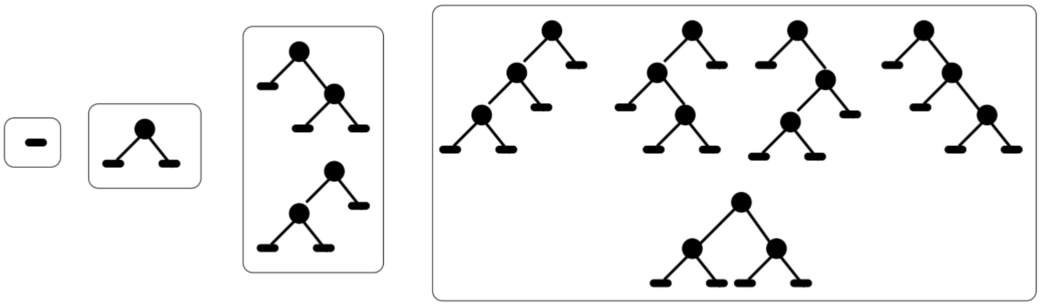
Dvojiška drevesa

V gramatiki jezika Haskell je dvojiško drevo T z vrednostmi v vozliščih:

$$data\ T\ a = None \mid Node\ a\ (T\ a)\ (T\ a). \tag{2.7}$$

V naši algebri tipov izraz (2.7) postane vsota dveh tipov: tipa *None*, ki je ekvivalenten (po definiciji 2.1) tipu *()*, in produkta tipov. Tokrat gre za produkt *treh* tipov, vendar je ta le gnezden produkt *(a, (b, c))*.
V naši algebri tipov izraz (2.7) zapišemo kot

$$T(a) = 1 + a \cdot T(a)^2, \tag{2.8}$$



Slika 2.2: Binarna drevesa

torej s kvadratno enačbo (2.8). Do njene rešitve bi se lahko dokopali podobno kot v računu (2.5) s substitucijo, a poskusimo lahko tudi z algebro tipov, kot smo storili v računu (2.6). Tedaj je rešitev kvadratne enačbe (2.8)

$$T(a) = \frac{1 - \sqrt{1 - 4 \cdot a}}{2 \cdot a}, \tag{2.9}$$

izraz, ki ga spet razumemo le kot način kodiranja informacije in ki ga razčlenimo v Taylorjevo vrsto

$$T(a) = 1 \cdot a^0 + 1 \cdot a + 2 \cdot a^2 + 5 \cdot a^3 + 14 \cdot a^4 + 42 \cdot a^5 \cdots \tag{2.10}$$

Zadnji izraz nam daje informacijo, da obstaja: *eno prazno* dvojiško drevo (člen $1 \cdot a^0$), *eno* dvojiško drevo *s po enim vozliščem* (člen $1 \cdot a$), *dve* dvojiški drevesi *s po dvema vozliščema* (člen $2 \cdot a^2$), *pet* dvojiških dreves *s po tremi vozlišči* (člen $5 \cdot a^3$) ..., kot je prikazano na sliki 2.2 in tudi v [12].

Izraz (2.10) torej prešteva različne vrste dvojiških dreves. Ta lastnost preštevanja nam je poznana iz kombinatoričnih rodov (ang. combinatiorial species) [19]. Vendar *rodovi* niso isto kot tipi; če enakosti s tipi brezskrbno manipuliramo, lahko naletimo na težave, kot bomo spoznali v naslednjem razdelku 2.1.5.

2.1.5 Algebraične manipulacije

Namen algebraičnega manipuliranja enakosti je, da bi izpeljali nova spoznanja. Vprašanje pa je, ali lahko s tipi računamo tako svobodno kot smo vajeni iz algebre števil, ne pa le, da neko enakost izrazimo v drugi obliki, kot smo enakost (2.8) izrazili kot (2.9). Odgovor je ne, razlogi pa sledeči.

Zaradi neobstoja negativnih tipov gre v primeru algebraičnih podatkovnih tipov za polkolobar (tj. kolobar brez inverzov). Zato moramo poznati podrobnosti ustreznih algebrskih struktur, da vemo, katere algebraične manipulacije na tipih so legalne (in kdaj). Vemo naslednje [11]:
Če pričnemo z enakostjo med kompleksnim številom $t \in \mathbb{C}$ in poljubnim polinomom $p(t)$ s kompleksnimi koeficienti

$$t = p(t),$$

in jo z algebraičnimi manipulacijami razvijemo v

$$q_1(t) = q_2(t),$$

kjer q_1 in q_2 nista konstanti, potem so te manipulacije legalne tudi v katerem-koli polkolobarju. To pomeni, da je rezultat legalen tudi za algebraične podatkovne tipe (saj ti tvorijo polkolobar) in da obstaja postopek, ki nas pripelje do identičnega rezultata brez uporabe inverzov (tj. deljenja in odštevanja).

Navedimo dva primera algebraičnih manipulacij. Iz enakosti (2.8) lahko z manipulacijami razvijemo enakost

$$T(a)^7 = T(a),$$

ki veli, da je sedmerec dvojiških dreves izomorfen enemu samemu dvojiškemu drevesu. Na prvi pogled to ni posebno globok rezultat, saj sta obe množici števno neskončni. A obstaja eksplicitna bijekcija med množicama, kot je pokazano v [5], kjer avtor pokaže, da za bijekcijo ni treba gledati globlje od štirih nivojev drevesa. Rezultat algebraičnih manipulacij je veljaven (legalen), saj obstaja izpeljava brez uporabe inverzov. A, če obe strani enačbe delimo s T, pridemo do enakosti

$$T(a)^6 = 1,$$

ki veli, da je šesterec dvojiških dreves izomorfen enoti (tj. *Unit*). To pa je očiten nesmisel in manipulacij ni mogoče predrugačiti v takšne, ki ne bi vsebovale inverzov.

V naštetih razlogih vidimo veliko pomanjkljivost algebre algebraičnih tipov: *pri manipuliranju z enakostmi zahteva dobro poznavanje zakonitosti algebrskih struktur.* Zato se bomo s tretjim poglavjem (Programski prostori) lotili konstrukcije modela, ki bo vseboval inverze in zato omogočal algebraične manipulacije, kot jih je bralec vajen iz algebre števil. Model bo slonel na paradigmi globokega učenja, ki jo spoznamo v naslednjem razdelku 2.2.

2.2 Globoko učenje in pomanjkanje formalizma

Globoko učenje je uporaba umetnih nevronskih mrež za učenje nalog. Nevronska mreža je definirana z rekurzivno enačbo

$$L^{i+1} = \phi_i(W_i \cdot L^i + b_i),$$

kjer je L^i izhod i-tega nivoja, W_i in b_i sta uteži in pristranost (ang. bias), operacija $\cdot$ pa bilinearna operacija (kot denimo tenzorska kontrakcija ali konvolucija), ϕ_i pa aktivacijska (ne-linearna) funkcija. Obstaja množica različnih arhitektur [21, 17, 16], ki so polne po Turingu, če omogočajo rekurzijo [14]. V zadnjih letih se je globoko učenje izkazalo na široki paleti problemov, kot so: prepoznava slik [28], sledenje [4], generiranje besedil, govora in slik [6], obravnavanje naravnega jezika [24], itd.

Globoko učenje temelji na multilinearnih transformacijah, ki se dajo zlahka paralelizirati (prva od Backusovih skrbi), kar s pridom izkoriščamo za implementacije na GPU [1]. Multilinearne transformacije lahko opišemo s tenzorsko algebro. To sicer omogoča algebraične manipulacije, a potrebni formalizem še ne obstaja v polni obliki [22].

2.2.1 Pomanjkanje formalizma

Globoko učenje je izrazljivo s tenzorsko algebro, a sklepanje o implementiranih modelih še vedno poteka v (matematičnem) jeziku, ki je ločen od implementacijskega jezika. Pravzaprav odsotnost formalizma sega tako daleč, da dandanes še ni natančne definicije globokega učenja. Leon Buttou, spoštovan strokovnjak s področja globokega učenja, je v korespondenci z avtorjem dejal:

> *"... There is no proper definition of what deep learning even is at this stage. Your [avtorjeva] theory of Operational Calculus on Programming Spaces [35] could offer a first such definition through the formalism you [avtor] propose."*
>
> – dr. Leon Buttou, *v korespondenci z avtorjem*

Snovanje tega manjkajočega formalizma je predmet našega dela. Konstruirali bomo algebraični jezik, ki predstavlja poln model globokega učenja. Z izpeljanim formalizmom bomo predlagali in prikazali nov način izražanja programov ter kako ta način uporabiti za izpeljavo novih arhitektur globokega učenja v teoretično utemeljenem kontekstu.

2.3 Tenzorska algebra

Tenzorska algebra vektorskega prostora V, ki jo označimo s $T(V)$, je algebra tenzorjev na V, pri kateri je bilinearna preslikava tenzorski produkt. Je prosta algebra (ang. free algebra) na V in hkrati najsplošnejša algebra, ki vsebuje V, v smislu korespondence z univerzalno lastnostjo [29]. Tenzorska algebra je pomembna, saj porodi mnoge algebre, kot kvocientne algebre $T(V)$, kar bomo s pridom izkoristili.

2.3.1 Univerzalna lastnost

Tenzorska algebra zadošča sledeči univerzalni lastnosti, ki formalno izraža izjavo, da je najsplošnejša algebra, ki vsebuje V.

Vsaka linearna transformacija $f : V \to A$, iz V v algebro A nad poljem K, je lahko enkratno razširjena na algebro morfizmov iz $T(V)$ v A, kot je prikazano na sledečem komutativnem diagramu

$$\begin{tikzcd} V \arrow[r, "i"] \arrow[dr, "f"'] & T(V) \arrow[d, dashed, "\tilde{f}"] \\ & A \end{tikzcd}$$

kjer je i kanonična vključitev V v $T(V)$. Definiramo lahko tenzorsko algebro $T(V)$ kot edino algebro, ki zadošča tej lastnosti, a kljub temu moramo pokazati, da tak objekt res obstaja [2].

2.3.2 Kvocienti

Zaradi splošnosti tenzorske algebre je mnoge druge algebre mogoče konstruirati tako, da pričnemo s tenzorsko algebro, v katero vpeljemo določene relacije nad generatorji, tj. konstruiramo določene kvocientne algebre $T(V)$. Primer tega je simetrična algebra, ki jo bomo uporabili tudi v tem delu.

2.4 Avtomatsko odvajanje

Avtomatsko odvajanje je množica tehnik za izračun odvodov funkcij, ki so implementirane kot računalniški programi. Izkorišča dejstvo, da vsak program, ne glede na svojo kompleksnost, sestoji iz izvedbe osnovnih operacij in funkcij. Tehnike omogočajo učinkovite izračune odvodov, v $\mathcal{O}(1)$-krat daljšem času [8].

2.4.1 Vnaprejšnji in vzvratni način avtomatskega odvajanja

Temelj avtomatskega odvajanja je dekompozicija diferencialov s pomočjo verižnega pravila. Navadno sta predstavljena dva načina avtomatskega odvajanja. *Vnaprejšnji način* [20] in *vzvratni način* [18] avtomatskega odvajanja: vnaprejšnji način določa obiskovanje verižnega pravila od navznoter proti navzven, medtem ko vzvratni način določa obisk od navzven proti navznoter.

V splošnem sta obe obliki specifična primera uporabe operatorja kompozicije programov, ki ga izpeljemo v tem delu (glej razdelek 5.2) in s tem avtomatsko odvajanje dvignemo iz učinkovitega načina izračuna odvodov v algebraično orodje.

3

Programski prostori

3.1 Programi kot preslikave vektorskih prostorov

Programe bomo modelirali kot preslikave vektorskih prostorov, vase. Če se osredotočimo na realne spremenljivke (tipa float ali double), je trenutno stanje pomnilnika predstavljivo z visoko-dimenzionalnim vektorjem. Množico vseh možnih stanj pomnilnika lahko modeliramo s končno-dimenzionalnim vektorskim prostorom $\mathcal{V} \simeq \mathbb{R}^n$ (podobno kot pri *nevronskih Turingovih strojih* [15]). Prostor $\mathcal{V}$ imenujemo *pomnilniški prostor programa.* Akcija (učinkovanje) programa P nad svojim pomnilniškim prostorom $\mathcal{V}$ zapišemo kot preslikavo

$$P : \mathcal{V} \to \mathcal{V}. \tag{3.1}$$

Programski prostor je prostor vseh preslikav $\mathcal{V} \to \mathcal{V}$, ki se dajo zapisati v obliki programa v izbranem programskem jeziku.

Definicija 3.1 (Evklidski stroj) *Par* $(\mathcal{V}, \mathcal{F})$*, kjer sta*

- $\mathcal{V}$ *končno-dimenzionalni vektorski prostor nad obsegom* K*,*
- $\mathcal{F} < \mathcal{V}^{\mathcal{V}}$ *podprostor prostora* $\mathcal{V}^{\mathcal{V}}$ *vseh preslikav* $\mathcal{V} \to \mathcal{V}$*,*

imenujemo Evklidski stroj *(z akcijami iz* $\mathcal{F}$ *nad* $\mathcal{V}$*).*

Na prvi pogled se Evklidski stroj zdi kot opis funkcijskega programiranja s kompozicijami, podedovanimi od $\mathcal{F}$. Vtis, da je Evklidski stroj videti kot model funkcijskega programiranja, je pravilen, saj želimo, da bi Evklidski stroj odražal in zajemal prednosti funkcijskega programiranja. Vendar bomo v naslednjem poglavju $\mathcal{F}$ dodatno omejili s pogojem, da so njeni elementi odvedljivi.

3.2 Odvedljive preslikave in programi

Pred definiranjem odvedljivih programov se spomnimo nekaj definicij.

Definicija 3.2 (Odvod) *Naj bosta V in U Banachova prostora. Preslikava $P : V \to U$ je odvedljiva v točki $\mathbf{x} \in V$, če obstaja omejen linearen operator $TP_{\mathbf{x}} : V \to U$ kjer*

$$\lim_{\mathbf{h}\to 0} \frac{\|P(\mathbf{x}+\mathbf{h}) - P(\mathbf{x}) - TP_{\mathbf{x}}(\mathbf{h})\|}{\|\mathbf{h}\|} = 0. \tag{3.2}$$

Preslikava $TP_{\mathbf{x}}$ se imenuje Fréchetov odvod *preslikave P v točki $\mathbf{x}$.*

Za preslikave $\mathbb{R}^n \to \mathbb{R}^m$ je Fréchetov odvod izrazljiv kot množenje vektorja $\mathbf{h}$ z matriko $\mathbf{J}_{P,\mathbf{x}}$ parcialnih odvodov komponent preslikave P:

$$T_{\mathbf{x}}P(\mathbf{h}) = \mathbf{J}_{P,\mathbf{x}} \cdot \mathbf{h}.$$

V nadaljevanju tega razdelka privzemamo, da je preslikava $P : V \to U$ odvedljiva za vse $\mathbf{x} \in V$. Odvod definira preslikavo iz V v množico omejenih linearnih preslikav iz V v U. Privzemamo tudi, da sta V in U končnodimenzionalna. Potem je prostor linearnih preslikav iz V v U izomorfen tenzorskem produktu $U \otimes V^*$; izomorfizem je podan s tenzorskimi kontrakcijami in preslika preprost tenzor $\mathbf{u} \otimes f \in U \otimes V^*$ v linearno preslikavo

$$\mathbf{u} \otimes f : \mathbf{x} \mapsto f(\mathbf{x}) \cdot \mathbf{u}. \tag{3.3}$$

Odvod definira preslikavo

$$\partial P \ : \ V \to U \otimes V^* \tag{3.4}$$

$$\partial P \ : \ \mathbf{x} \mapsto T_{\mathbf{x}}P. \tag{3.5}$$

Obravnavamo pa lahko tudi odvedljivost samega odvoda ∂P, kar vodi do višjih odvodov.

Definicija 3.3 (Višji odvodi) *Naj bo $P : V \to U$ preslikava iz vektorskega prostora V v vektorski prostor U. Odvod $\partial^k P$ reda k preslikave P je preslikava*

$$\partial^k P \quad : \quad V \to U \otimes (V^*)^{\otimes k} \tag{3.6}$$

$$\partial^k P \quad : \quad \mathbf{x} \mapsto T_{\mathbf{x}} \left(\partial^{k-1} P\right) \tag{3.7}$$

Opomba 3.1 *V zgornji definiciji predpostavljamo, da je preslikava P, kot tudi vsi njeni odvodi, odvedljiva za vse točke $\mathbf{x} \in \mathcal{V}$. V nasprotnem bi zgornje definicije kljub temu lahko izrazili lokalno, kar bi prineslo v večini tehnične težave.*

Naj bo $\mathbf{e}_1, \ldots, \mathbf{e}_n$ baza prostora U in $x_1, \ldots, x_m$ baza prostora V^*. Z izrazom $P_i = x_i \circ P$ označimo i-to komponento preslikave P glede na bazo $\{\mathbf{e}_i\}$. Potem lahko definiramo $\partial^k P$ s smernimi (parcialnimi) odvodi:

$$\partial^k P = \sum_{\forall_{i,\alpha}} \frac{\partial^k P_i}{\partial x_{\alpha_1} \ldots \partial x_{\alpha_k}} \mathbf{e}_i \otimes dx_{\alpha_1} \otimes \ldots \otimes dx_{\alpha_k}. \tag{3.8}$$

3.3 Odvedljivi programi

Želimo si, da bi tudi odvode programov v Evklidskem stroju lahko predstavili kot programe v istem Evklidskem stroju. S tem namenom bomo definirali tri podprostore v pomnilniškem prostoru $\mathcal{V}$, ki bodo opisovali, kako različni deli pomnilniškega prostora vplivajo na končni rezultat programa.

Označimo z $\mathbf{e}_1, \ldots, \mathbf{e}_n$ standardno bazo pomnilniškega prostora $\mathcal{V}$ in z $x_1, \ldots x_n$ dualno bazo $\mathcal{V}^*$. Funkcije x_i so koordinatne funkcije na $\mathcal{V}$ in ustrezajo posamičnim lokacijam (spremenljivkam) v pomnilniku programa.

Definicija 3.4 *Za vsak program P programskega prostora $\mathcal{F} < \mathcal{V}^{\mathcal{V}}$ definiramo* vhodni prostor $I_P < \mathcal{V}$ *(prostor parametrov) in* zunanji prostor $O_P < \mathcal{V}$

kot minimalna podprostora, ki ju napenja standardna baza, na tak način, da je preslikava P_e, definirana s komutativnim diagramom

$$\begin{tikzcd} \mathcal{V} \arrow[r, "P"] & \mathcal{V} \arrow[d, "\mathrm{pr}_{O_P}"] \\ I_P \arrow[u, hook, "\vec{i} \mapsto \vec{i} + \vec{f}"] \arrow[r, "P_e"] & O_P \end{tikzcd} \tag{3.9}$$

neodvisna od izbire elementa $\vec{f} \in F_P = (I_P + O_P)^{\perp}$.

Prostor $F_P = (I_P + O_P)^{\perp}$ *poimenujemo* prost prostor *programa* P.

Spremenljivke x_i, ki ustrezajo standardni bazi vektorjev in napenjajo parametre, izhode ter prost prostor, so imenovane *vhodne spremenljivke*, *izhodne spremenljivke* in *proste spremenljivke. Proste spremenljivke* so tiste, ki jih akcija programa ne naslovi in nimajo vpliva na končni rezultat. Izhod programa je odvisen zgolj od vrednosti vhodnih spremenljivk in sestoji iz spremenljivk, katerih vrednosti so bile spremenjene med izvajanjem programa. Vhodne in izhodne spremenljivke imajo lahko neprazen presek.

Preslikavo P_e (3.9) imenujemo *akcijska preslikava programa* P in opisuje dejansko akcijo programa P nad svojim pomnilniškim prostorom $\mathcal{V}$, pri čemer P_e ignorira prost prostor programa.

Ko govorimo o odvedljivosti programov, nas bo predvsem zanimal odvod akcijske preslikave.

Definicija 3.5 (Avtomatsko odvedljivi programi) *Program* $P_{\mathcal{V}} \to \mathcal{V}$ *je* avtomatsko odvedljiv, *če obstaja vložitev prostora* $O_P \otimes I_P^*$ *v prost prostor* F_P *in tak program* $(1+\partial)P : \mathcal{V} \to \mathcal{V} \oplus (\mathcal{V} \otimes \mathcal{V}^*)$, *da je njegova akcijska preslikava*

$$P_e \oplus \partial P_e : I_P \to O_P \oplus (O_P \otimes I^*). \tag{3.10}$$

Program $P : \mathcal{V} \to \mathcal{V}$ *je* avtomatsko odvedljiv do reda k, *če obstaja tak program* $(1 + \partial + \cdots + \partial^k)P : \mathcal{V} \to \mathcal{V} \oplus (\mathcal{V} \otimes \mathcal{V}^*) \oplus \cdots \oplus (\mathcal{V} \otimes \mathcal{V}^{k*})$, *da je njegova akcijska preslikava*

$$P_e \oplus \partial P_e \oplus \ldots \partial^k P_e : I_P \to O_P \oplus (O_P \otimes I^*) \oplus \cdots \oplus \left(O_P \otimes \left(I_p^*\right)^{k\otimes}\right). \tag{3.11}$$

Če je program $P : \mathcal{V} \to \mathcal{V}$ avtomatsko odvedljiv, potem je tudi odvedljiv kot preslikava $\mathcal{V} \to \mathcal{V}$. Toda samo odvod akcijske preslikave programa P lahko implementiramo kot program, saj je v praksi pomnilniški prostor omejen na $\mathcal{V}$, da pa lahko odvajamo program do reda k, moramo izračunati in shraniti vse odvode reda k ter manj.

4

Odvedljivi programski prostori

Pomnilniški prostor programa je redko obravnavan kot kaj več kot zgolj shramba. Da pa bi obogatili Evklidski stroj z dodano strukturo, se osredotočimo prav nanj. Ohlapno rečeno je funkcijsko programiranje opisano z monoidi (grupami brez inverzov), zato se (multi)linearno algebraičen opis pomnilnika zdi primeren korak k pridobitvi dodatne strukture.

4.1 Navidezni pomnilniški prostor

Motivirani z Definicijo 3.5 definiramo navidezni pomnilniški prostor programov kot zaporedje vektorskih prostorov po sledeči rekurzivni formuli:

$$\mathcal{V}_0 = \mathcal{V} \tag{4.1}$$

$$\mathcal{V}_k = \mathcal{V}_{k-1} + (\mathcal{V}_{k-1} \otimes \mathcal{V}^*) \,. \tag{4.2}$$

Poudarjamo, da vsota ni direktna, saj so nekateri podprostori prostorov $\mathcal{V}_{k-1}$ in $\mathcal{V}_{k-1} \otimes \mathcal{V}^*$ izomorfni in bodo med seboj izenačeni.

Prostor, ki zadošča rekurzivni formuli (4.2), je

$$\mathcal{V}_k = \mathcal{V} \otimes \left(K \oplus \mathcal{V}^* \oplus (\mathcal{V}^* \otimes \mathcal{V}^*) \oplus \cdots \oplus (\mathcal{V}^*)^{\otimes k}\right) = \mathcal{V} \otimes T_k(\mathcal{V}^*), \tag{4.3}$$

kjer je $T_k(\mathcal{V}^*)$ podprostor tenzorske algebre $T(\mathcal{V}^*)$, ki sestoji iz linearnih kombinacij tenzorjev ranga k ali manj. Ta konstrukcija nam omogoča, da definiramo vse odvode kot preslikave z domeno $\mathcal{V}$ in ko-domeno $\mathcal{V} \otimes T(V^*)$.

Kot tak je navidezni pomnilniški prostor preslikava iz sebe vase,

$$\mathcal{V}_n : \mathcal{V} \to \mathcal{V}, \tag{4.4}$$

ki je za poljuben $\mathbf{W} \in \mathcal{V}_n$ definirana kot,

$$\mathbf{W}(\mathbf{v}) = \mathbf{w}_0 + \mathbf{w}_1 \cdot \mathbf{v} + \cdots + \mathbf{w}_n \cdot (\mathbf{v})^{\otimes n}, \tag{4.5}$$

vsota večih kontrakcij (kjer so $\mathbf{w}_i \in \mathcal{V}_i$). Enakost (4.5) bomo strogo definirali v razdelku 5.1. S takšno konstrukcijo dobijo razširitve in kontrakcije pomnilniškega prostora (ki spominjajo na dihanje sklada) širši pomen kot zgolj hranjenje podatkov. Prav to pa motivira novo definicijo.

Definicija 4.1 (Navidezni pomnilniški prostor) *Naj bo par* $(\mathcal{V}, \mathcal{F})$ *Evklidski stroj in*

$$\mathcal{V}_\infty = \mathcal{V} \otimes T(\mathcal{V}^*) = \mathcal{V} \oplus (\mathcal{V} \otimes \mathcal{V}^*) \oplus \ldots, \tag{4.6}$$

kjer je $T(\mathcal{V}^*)$ *tenzorska algebra dualnega prostora* $\mathcal{V}^*$*. Potem* $\mathcal{V}_\infty$ *poimenujemo* navidezni pomnilniški prostor *Evklidskega stroja* $(\mathcal{V}, \mathcal{F})$*.*

Izraz *navidezni pomnilnik* smo uporabili zato, ker je možno v pomnilniški prostor $\mathcal{V}$ vložiti le omejeno število podprostorov $\mathcal{V}_\infty$, kar velja tudi za navidezni pomnilnik v računalnikih.

Vsak program $P : \mathcal{V} \to \mathcal{V}$ lahko razširimo na preslikavo nad navideznim pomnilniškim prostorom $\mathcal{V}_\infty$, tako da nastavimo prvo komponento direktne vsote (4.6) na P, ostale komponente pa na 0. Podobno lahko odvode $\partial^k P$ obravnavamo kot preslikave iz $\mathcal{V}$ v $\mathcal{V}_\infty$, tako da nastavimo k-to komponento direktne vsote (4.6) na $\partial^k P$, ostale pa na 0.

4.2 Odvedljivi programski prostori

Za jasnost izražanja definiramo sledeče funkcijske prostore:

$$\mathcal{F}_n = \{f : \mathcal{V} \to \mathcal{V} \otimes T_n(\mathcal{V}^*)\} \tag{4.7}$$

Te prostore lahko gledamo kot podprostore $\mathcal{F}_\infty = \{f : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)\}$, saj je $\mathcal{V}$ naravno vložen v $\mathcal{V} \otimes T(\mathcal{V}^*)$. Fréchetov odvod definira operator na prostoru gladkih preslikav $\mathcal{F}_\infty$. Slika katerekoli preslikave $P : \mathcal{V} \to \mathcal{V}$ operatorja ∂ je njen prvi odvod, višji odvodi pa so kar potence operatorja ∂, aplicirane na P. Tako je ∂^k preslikava med funkcijskimi prostori (4.7),

$$\partial^k : \mathcal{F}^n \to \mathcal{F}^{n+k}. \tag{4.8}$$

Definicija 4.2 (Odvedljiv programski prostor) Odvedljiv programski prostor $\mathcal{P}_0$ *je vsak podprostor prostora* $\mathcal{F}_0$*, kjer*

$$\partial \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*). \tag{4.9}$$

Prostor $\mathcal{P}_n < \mathcal{F}_n$*, ki ga napenja* $\{\partial^k \mathcal{P}_0; \quad 0 \leq k \leq n\}$ *nad* K*, imenujemo* odvedljiv programski prostor reda n. *Ko so vsi elementi* $\mathcal{P}_0$ *analitični,* $\mathcal{P}_0$ *imenujemo* analitični programski prostor.

Definicijo odvedljivih programskih prostorov višjega reda upravičimo z naslednjim izrekom.

Izrek 4.1 (Neskončna odvedljivost) *Vsak odvedljiv programski prostor* $\mathcal{P}_0$ *je* neskončnokrat odvedljiv programski prostor*, kar pomeni,*

$$\partial^k \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{4.10}$$

za vsak $k \in \mathbb{N}$.

Dokaz. Z indukcijo na redu k. Za $k = 1$ izrek drži po definiciji. Predpostavimo $\forall_{P \in \mathcal{P}_0}, \partial^n \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*)$. S $P^i_{\alpha,k}$ označimo komponento k-tega odvoda, kjer multi-indeks α označuje komponento $T(\mathcal{V}^*)$ in indeks i komponento $\mathcal{V}$.

$$\partial^{n+1} P^i_{\alpha,k} = \partial(\partial^n P^i_\alpha)_k \wedge (\partial^n P^i_\alpha) \in \mathcal{P}_0 \implies \partial(\partial^n P^i_\alpha)_k \in \mathcal{P}_0 \otimes T(\mathcal{V}^*) \tag{4.11}$$

$$\implies$$

$$\partial^{n+1} \mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*)$$

Tako po indukciji trditev drži za vsak $k \in \mathbb{N}$. □

Posledica 4.1 *Odvedljiv programski prostor reda n, $\mathcal{P}_n : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$, lahko vložimo v tenzorski produkt funkcijskega prostora $\mathcal{P}_0$ in prostora $T_n(\mathcal{V}^*)$ multi-tenzorjev reda n ali manj:*

$$\mathcal{P}_n < \mathcal{P}_0 \otimes T_n(\mathcal{V}^*). \tag{4.12}$$

V limiti $n \to \infty$ obravnavamo

$$\mathcal{P}_\infty < \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*), \tag{4.13}$$

kjer je $\mathcal{T}(\mathcal{V}^*) = \prod_{k=0}^{\infty}(\mathcal{V}^*)^{\otimes k}$ *algebra tenzorskih vrst*, tj. algebra neskončnih formalnih tenzorskih vrst.[1]

4.3 Navidezni tenzorski stroji

V tem razdelku bomo predlagali abstrakten računski model, ki bo konstruiral odvedljive programske prostore. Takšen model bo omogočal analitično preiskovanje programov z algebraičnimi prijemi.

Iz Izreka 4.1 sledi, da par $(\mathcal{V}, \mathcal{P}_0)$ – skupaj s strukturo tenzorske algebre $T(\mathcal{V}^*)$ – zadošča za konstrukcijo odvedljivih programskih prostorov $\mathcal{P}_\infty$ z uporabo linearne kombinacije elementov $\mathcal{P}_0 \otimes T(\mathcal{V}^*)$. To motivira naslednjo definicijo.

Definicija 4.3 (Navidezni tenzorski stroj) *Par* $(\mathcal{V}, \mathcal{P}_0)$ *je* navidezni tenzorski stroj, *če je*

- *$\mathcal{V}$ končno-dimenzionalni vektorski prostor,*
- *$\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ navidezni pomnilniški prostor,*
- *$\mathcal{P}_0$ analitični programski prostor nad $\mathcal{V}$.*

Ko komponiramo kontrakcije (4.5) pomnilnika z aktivacijskimi funkcijami $\phi \in \mathcal{P}$, opazimo, da so tenzorske mreže [35]

$$\mathcal{N}(v) = \phi_k \circ W_k \circ \cdots \circ \phi_0 \circ W_0(v), \tag{4.14}$$

[1] Algebra tenzorskih vrst je dopolnitev tenzorske algebre $T(\mathcal{V}^*)$ v ustrezni topologiji.

osnovni programi navideznih tenzorskih strojev (standardne nevronske mreže pridobimo s pogojem $\forall_i(W_i \in \mathcal{V}_1)$). Formulacijo (4.14) lahko posplošimo na konvolucijske modele, vendar tega v tem delu ne bomo razložili.

Opomba 4.1 (TensorFlow) *Knjižnice kot je TensorFlow [9], lahko razumemo kot implementacije navideznih tenzorskih strojev, ki vsebujejo določene podmnožice odvedljivih programskih prostorov* $\mathcal{P}_1 < \mathcal{P}_0 \otimes T(\mathcal{V}^*)$.

4.4 Implementacija

Odprtokodna implementacija razširitve pomnilniškega prostora $\mathcal{V}$ na navidezni pomnilniški prostor $\mathcal{V} \otimes T(\mathcal{V}^*)$ (ki služi kot algebra programov) in odvedljivega programskega prostora (odvedljivega programskega jezika C++)

$$\sum_{i=0}^{n} \partial^i C\text{++} < \mathcal{P}_n : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*) \tag{4.15}$$

je na razpolago na avtorjevi strani [32]. Implementacijo spremljata članek [34], kjer smo ubesedili proces implementacije teorije tega dela, in učno besedilo uporabe knjižnice [33].

5

Operatorski račun nad programskimi prostori

Po Posledici 4.1 lahko izračun odvodov preslikav $P : \mathcal{V} \to \mathcal{V}$ predstavimo z enim slikanjem τ. Operator τ_n definiramo kot direktno vsoto operatorjev

$$\tau_n = 1 + \partial + \partial^2 + \cdots + \partial^n \tag{5.1}$$

Slika $\tau_n P(\mathbf{x})$ je multi-tenzor ranga k, ki je direktna vsota vrednosti preslikave in vseh odvodov reda $n \leq k$ v točki $\mathbf{x}$:

$$\tau_k P(\mathbf{x}) = P(\mathbf{x}) + \partial_{\mathbf{x}} P(\mathbf{x}) + \partial_{\mathbf{x}}^2 P(\mathbf{x}) + \cdots + \partial_{\mathbf{x}}^k P(\mathbf{x}). \tag{5.2}$$

Tako operator τ zadošča rekurzivni relaciji

$$\tau_{k+1} = 1 + \partial \tau_k, \tag{5.3}$$

ki je lahko uporabljena za rekurzivno konstrukcijo programskih prostorov poljubnega reda.

Trditev 5.1 *Za konstrukcijo $\mathcal{P}_n$ iz $\mathcal{P}_0$ je potrebno zgolj eksplicitno poznavanje operatorja $\tau_1 : \mathcal{P}_0 \to \mathcal{P}_1$.*

Dokaz. Konstrukcijo dosežemo skozi argument (4.11) dokaza Izreka 4.1, ki omogoča enostavno implementacijo po nareku (5.3). □

Opomba 5.1 *Preslikave* $\mathcal{V} \otimes T(\mathcal{V}^*) \to \mathcal{V} \otimes T(\mathcal{V}^*)$ *lahko konstruiramo s tenzorsko algebro in kompozicijami programov v* $\mathcal{P}_n$.

Definicija 5.1 (Produkt algebre) *Za vsako bilinearno preslikavo*

$$\cdot : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$$

lahko definiramo bilinearen produkt $\cdot$ *na* $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$, *tako da sledimo pravilu za preproste tenzorje:*

$$(\mathbf{v} \otimes f_1 \otimes \dots f_k) \cdot (\mathbf{u} \otimes g_1 \otimes \dots g_l) = (\mathbf{v} \cdot \mathbf{u}) \otimes f_1 \otimes \dots f_k \otimes g_1 \otimes \dots g_l \tag{5.4}$$

ki se linearno razširi na celoten prostor $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$.

Izrek 5.1 (Algebra programov) *Za vsako bilinearno preslikavo* $\cdot : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ *je neskončnokrat odvedljiv programski prostor* $\mathcal{P}_\infty$ *funkcijska algebra s produktom definiranim v* (5.4).

5.1 Razvoj v tenzorsko vrsto

Naslednje, kar naš model potrebuje, je operator, ki bi prestavil vrednost funkcije iz njene začetne vrednosti. Tak operator bi lahko kasneje uporabili za implementacijo iteratorjev in komponerjev.

Takšen operator definiramo kot

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{(h\partial)^n}{n!}$$

V koordinatah lahko operator $e^{h\partial}$ zapišemo kot vrsto po multi-indeksih α

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall_{i,\alpha}} \frac{\partial^n}{\partial x_{\alpha_1} \dots \partial x_{\alpha_n}} \mathbf{e}_i \otimes dx_{\alpha_1} \otimes \dots \otimes dx_{\alpha_n}. \tag{5.5}$$

Torej je operator $e^{h\partial}$ preslikava med funkcijskimi prostori (4.7)

$$e^{h\partial} : \mathcal{P} \to \mathcal{P}_\infty.$$

Hkrati pa definira preslikavo

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*) \tag{5.6}$$

tako, da vzame sliko preslikave $e^{h\partial}(P)$ v določeni točki $\mathbf{x} \in \mathcal{V}$. Z uporabo (5.6) lahko konstruiramo preslikavo iz prostora programov v prostor polinomov. Poudarjamo, da je prostor polinomov $\mathcal{V} \to K$ izomorfen simetrični algebri $S(\mathcal{V})$, ki pa je kvocient tenzorske algebre $T(\mathcal{V}^*)$. Vsakemu elementu v $\mathcal{V} \otimes T(\mathcal{V}^*)$ lahko priredimo pripadajoč element iz $\mathcal{V} \otimes S(\mathcal{V}^*)$, in sicer polinomsko preslikavo $\mathcal{V} \to \mathcal{V}$. Zatorej, podobno kot pri (4.13), obravnavamo dopolnitev simetrične algebre $S(\mathcal{V}^*)$ kot formalne potenčne vrste $\mathcal{S}(\mathcal{V}^*)$, ki so izomorfne kvocientu algebre tenzorskih vrst $\mathcal{T}(\mathcal{V}^*)$. To nas privede do

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^*). \tag{5.7}$$

Za vsak element $\mathbf{v}_0 \in \mathcal{V}$ je izraz $e^{h\partial}(\cdot, \mathbf{v}_0)$ preslikava $\mathcal{P} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^*)$, ki preslika program v formalno potenčno vrsto.

Korespondenco med multi-tenzorji v $\mathcal{V} \otimes T(\mathcal{V}^*)$ in polinomskimi preslikavami $\mathcal{V} \to \mathcal{V}$ lahko izrazimo z več kontrakcijami po vseh indeksih. Za preproste tenzorje $\mathbf{u} \otimes f_1 \otimes \ldots \otimes f_n \in \mathcal{V} \otimes (\mathcal{V}^*)^{\otimes n}$ je kontrakcija z $\mathbf{v} \in \mathcal{V}$ dana z aplikacijo ko-vektorja f_n na $\mathbf{v}$,[1]

$$\mathbf{u} \otimes f_1 \otimes \ldots \otimes f_n \cdot \mathbf{v} = f_n(\mathbf{v}) \mathbf{u} \otimes f_1 \otimes \ldots f_{n-1}. \tag{5.8}$$

Z večkratno uporabo kontrakcije preprostemu tenzorju pripišemo monomsko preslikavo z

$$\mathbf{u} \otimes f_1 \otimes \ldots \otimes f_n \cdot (\mathbf{v})^{\otimes n} = f_n(\mathbf{v}) f_{n-1}(\mathbf{v}) \cdots f_1(\mathbf{v}) \mathbf{u}. \tag{5.9}$$

Obe kontrakciji, (5.8) in (5.9), sta preko linearnosti razširljivi na prostore $\mathcal{V} \otimes (\mathcal{V})^{\otimes n}$ in dalje na $\mathcal{V} \otimes T(\mathcal{V}^*)$.[2]

[1] Za tenzorje drugega reda $\mathcal{V} \otimes \mathcal{V}^*$ kontrakcija korespondira z množenjem vektorja z matriko.

[2] Opazimo, da na preprostem tenzorju ranga ena $\mathbf{u} \in \mathcal{V}$ ne moremo uporabiti kontrakcije z $\mathbf{v}$. Za konsistentnost definiramo $\mathbf{u} \cdot \mathbf{v} = \mathbf{u}$ in k tenzorju reda nič, $\mathbf{u}$, pripišemo konstantno preslikavo $\mathbf{v} \mapsto \mathbf{u}$. Razširitev (5.9) v $\mathcal{V} \otimes T(\mathcal{V}^*)$ je lahko videna kot posplošitev afine preslikave, kjer tenzor ranga nič ustreza translaciji.

Za multi-tenzor $\mathbf{W} = \mathbf{w}_0 + \mathbf{w}_1 + \cdots + \mathbf{w}_n \in \mathcal{V} \otimes T_n(\mathcal{V}^*)$, kjer je $\mathbf{w}_k \in \mathcal{V} \otimes (\mathcal{V}^*)^{\otimes n}$, se večkratna kontrakcija z vektorjem $\mathbf{v} \in \mathcal{V}$ izraža kot polinomska preslikava

$$\mathbf{W}(\mathbf{v}) = \mathbf{w}_0 + \mathbf{w}_1 \cdot \mathbf{v} + \cdots + \mathbf{w}_n \cdot (\mathbf{v})^{\otimes n}. \tag{5.10}$$

Izrek 5.2 (Razvoj v tenzorsko vrsto) *Za program $P \in \mathcal{P}$ se razvoj v neskončno tenzorsko vrsto v točki $\mathbf{v}_0 \in \mathcal{V}$ izraža z več kontrakcijami*

$$\begin{aligned} P(\mathbf{v}_0 + h\mathbf{v}) = \Big((e^{h\partial} P)(\mathbf{v}_0)\Big)(\mathbf{v}) &= \sum_{n=0}^{\infty} \frac{h^n}{n!} \partial^n P(\mathbf{v}_0) \cdot (\mathbf{v}^{\otimes n}) \\ &= \sum_{n=0}^{\infty} \frac{h^n}{n!} \sum_{\forall i,\alpha} \frac{\partial^n P_i}{\partial x_{\alpha_1} \ldots \partial x_{\alpha_n}} \mathbf{e}_i \cdot dx_{\alpha_1}(\mathbf{v}) \cdot \ldots \cdot dx_{\alpha_n}(\mathbf{v}). \end{aligned} \tag{5.11}$$

Dokaz. Pokazali bomo, da $\frac{d^n}{dh^n}(\mathrm{LHS})|_{h=0} = \frac{d^n}{dh^n}(\mathrm{RHS})|_{h=0}$. Potem imata LHS in RHS kot funkciji h enaki Taylorjevi vrsti in sta posledično enaki.

$\Longrightarrow$

$$\frac{d^n}{dh^n} P(\mathbf{v}_0 + h\mathbf{v})\Big|_{h=0} = \partial^n P(\mathbf{v}_0)(\mathbf{v})$$

$\Longleftarrow$

$$\frac{d^n}{dh^n} \left((e^{h\partial})(P)(\mathbf{v}_0)\right)(\mathbf{v})\Big|_{h=0} = \left((\partial^n e^{h\partial})(P)(\mathbf{v}_0)\right)(\mathbf{v})\big|_{h=0}$$

$$\wedge$$

$$\partial^n e^{h\partial}\big|_{h=0} = \sum_{i=0}^{\infty} \frac{h^i \partial^{i+n}}{i!}\Big|_{h=0} = \partial^n$$

$$\Longrightarrow$$

$$(\partial^n(P)(\mathbf{v}_0)) \cdot (\mathbf{v}^{\otimes n})$$

$\square$

Opomba 5.2 (Konvolucijske vrste) *Izrek 5.2 je mogoče posplošiti na konvolucije z uporabo Volterrovih vrst. S tem model vključuje tudi konvolucijske nevronske arhitekture.*

Iz zgornjega teorema trivialno sledi, da je operator $e^{h\partial}$ avtomorfizem algebre programov $\mathcal{P}_\infty$,

$$e^{h\partial}(p_1 \cdot p_2) = e^{h\partial}(p_1) \cdot e^{h\partial}(p_2) \tag{5.12}$$

kjer $\cdot$ predstavlja bilinearno preslikavo.

Opomba 5.3 (Posplošen operator premika) *Operator* $e^{h\partial}$ (5.6)*, evaluiran v točki* $h = 1$*, je posplošitev operatorja premika [31]. S teorijo, ki jo predstavimo v tem razdelku, pa operator premika v našem modelu omogoča več kot zgolj implementacijo iteratorjev, kot bo postalo jasno v prihajajočih razdelkih.*

Za specifičen $\mathbf{v}_0 \in \mathcal{V}$ posplošen operator premika označimo z

$$e^{\partial}|_{\mathbf{v}_0} : \mathcal{P} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*).$$

Ko je izbira $\mathbf{v}_0 \in \mathcal{V}$ arbitrarna, ga v zapisu izpuščamo.

S tem smo razvili operator premika, s katerim zadostimo potrebi, ki smo jo izrazili na začetku razdelka. Zdaj pa se obračamo k drugim zmožnostim, ki nam jih ta operator ponuja, in sicer zmožnostim algebraičnega sklepanja o analitičnih lastnostih programov.

Opomba 5.4 *Posledica* 4.1 *skozi* (4.13) *implicira* $e^{h\partial}(\mathcal{P}_0) \subset \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*)$*, kar omogoča učinkovito implementacijo v navideznem tenzorskem stroju s pomočjo operatorja* τ*.*

5.2 Operator kompozicije programov

V tem razdelku posplošimo tako *vnaprejšnji način* [20] kot tudi *vzvratni način* [18] avtomatskega odvajanja na poljubni red z enim samim operatorjem. Nato pa bomo demonstrirali, da naša teorija omogoča izračune na samih operatorjih, preden operator apliciramo na konkretni program, kar nadomesti kompleksne pojme s preprostimi izrazi.

Izrek 5.3 (Operator kompozicije programov) *Kompozicijo dveh programov $f \circ g \in \mathcal{P}$ lahko izrazimo kot*

$$e^{h\partial}(f \circ g) = \exp(\partial_f e^{h\partial_g})(g, f), \tag{5.13}$$

kjer je $\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \times \mathcal{P} \to \mathcal{P}_\infty$ operator na parih preslikav (g, f), pri čemer je ∂_g diferencialni operator, ki ga apliciramo na prvo komponento g, in ∂_f na drugo komponento f.

Dokaz. Pokazali bomo $\frac{d^n}{dh^n}(\text{LHS})|_{h=0} = \frac{d^n}{dh^n}(\text{RHS})|_{h=0}$. Potem imata LHS in RHS kot funkciji h enaki Taylorjevi vrsti in sta posledično enaki.

$\Longrightarrow$

$$\lim_{\|h\| \to 0} (\frac{d}{dh})^n e^{\partial}(f \circ g) = \lim_{\|h\| \to 0} \partial^n e^{h\partial}(f \circ g)$$

$$\Longrightarrow$$

$$\partial^n(f \circ g) \tag{5.14}$$

$\Longleftarrow$

$$\exp(\partial_f e^{h\partial_g}) = \exp\left(\partial_f \sum_{i=0}^{\infty} \frac{(h\partial_g)^i}{i!}\right) = \prod_{i=1}^{\infty} e^{\partial_f \frac{(h\partial_g)^i}{i!}} \left(e^{\partial_f}\right)$$

$$\Longrightarrow$$

$$\exp(\partial_f e^{h\partial_g})(g, f) = \sum_{\forall_n} h^n \sum_{\lambda(n)} \prod_{k \cdot l \in \lambda} \left(\frac{\partial_f \partial_g^l(g)}{l!}\right)^k \frac{1}{k!} \left(\left(e^{\partial_f}\right) f\right)$$

kjer $\lambda(n)$ predstavlja particije n. Torej

$$\lim_{\|h\| \to 0} (\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k \cdot l \in \lambda} \left(\frac{\partial_f \partial_g^l(g)}{l!}\right)^k \frac{1}{k!} \left(\left(e^{\partial_f}\right) f\right) \tag{5.15}$$

upoštevajoč dejstvo, da je evaluiranje $e^{\partial_f}(f)$ v točki $\mathbf{v} \in \mathcal{V}$ enako kot evaluiranje f v $\mathbf{v}$, je izraz (5.15) enak (5.14) po Faà di Bruno formuli.

$$\lim_{\|h\| \to 0} (\frac{d}{dh})^n \exp(\partial_f e^{h\partial_g}) = \sum_{\lambda(n)} n! \prod_{k \cdot l \in \lambda} \left(\frac{\partial_f \partial_g^l(g(v))}{l!}\right)^k \frac{1}{k!} \left(f(g(\mathbf{v}))\right) \tag{5.16}$$

□

Izrek 5.3 omogoča invariantno implementacijo operatorja kompozicije programov v $\mathcal{P}_n$, izraženo v tenzorski vrsti skozi (5.13) in (5.15).

S fiksacijo druge preslikave g v

$$\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \times \mathcal{P} \to \mathcal{P}_\infty, \tag{5.17}$$

je operator

$$\exp(\partial_f e^{h\partial_g})(\cdot, g) = g^* \left(e^{h\partial}\right) \tag{5.18}$$

povlek posplošenega operatorja premika $e^{h\partial}$ skozi g. S fiksacijo druge preslikave f v (5.17) pa je operator

$$\exp(\partial_f e^{h\partial_g})(f, \cdot) = f_* \left(e^{h\partial}\right) \tag{5.19}$$

potisk posplošenega operatorja premika $e^{h\partial}$ skozi f.

Opomba 5.5 (Poenoteno avtomatsko odvajanje) *Če je program izrazljiv kot $P = P_n \circ \dots P_1$, je uporaba operatorjev $\exp(\partial_f e^{h\partial_g})(\cdot, P_i)$ od $i = 1$ do $i = n$ ekvivalentno* vnaprejšnjemu načinu avtomatskega odvajanja. *Uporaba operatorjev $\exp(\partial_f e^{h\partial_g})(P_{n-i+1}, \cdot)$ v vzvratnem vrstnem redu pa je ekvivalentna* vzvratnem načinu avtomatskega odvajanja. *S tem oba načina posplošimo na poljubni red z enim samim operatorjem* (5.17)*, tako da fiksiramo primerno od obeh preslikav, ali f ali g.*

Torej operator (5.18) skozi (5.13) omogoča invariantnost od točke stopnje izvajanja programa, kar je pomembno pri dokazovanju pravilnosti programov. To je analogno principu splošne kovariance [23] v teoriji relativnosti, ki je invarianca oblike fizikalnih zakonov pod arbitrarnimi odvedljivimi transformacijami.

Posledica 5.1 *Operator $e^{h\partial}$ komutira s kompozicijami preko $\mathcal{P}$*

$$e^{h\partial}(p_2 \circ p_1) = e^{h\partial}(p_2) \circ e^{h\partial}(p_1).$$

Dokaz. Sledi iz (5.7) in Izreka 5.3. □

Takšne izračune pa lahko olajšamo tako, da jih izvedemo na samih operatorjih in se s tem izognemo manipulacijam tenzorskih vrst. Tako operatorji postanejo oblika abstrakcije.

Odvod $\frac{d}{dh}$ operatorja (5.18) je

$$\frac{d}{dh}\exp(\partial_f e^{h\partial_g})(g) = \partial_f(\partial_g g)e^{h\partial_g}\exp(\partial_f e^{h\partial_g})(g). \tag{5.20}$$

Opozarjamo pa na pomembno razliko z operatorjem $e^{h\partial_g}$, kateregা odvod je

$$\frac{d}{dh}e^{h\partial_g} = \partial_g e^{h\partial_g}. \tag{5.21}$$

S tema enakostma lahko poljubnokrat odvajamo operator povleka.

5.3 Primer izračuna na operatorjih

V primer pouka vzemimo izračun drugega odvoda operatorja (5.13)

$$\left(\frac{d}{dh}\right)^2 \exp\left(\partial_f e^{h\partial_g}\right)(g) = \frac{d}{dh}\left(\partial_f(\partial_g g)e^{h\partial_g}\exp\left(\partial_f e^{h\partial_g}\right)(g)\right),$$

ki je po enačbah (5.20) in (5.21) z uporabo algebre ter ustreznih aplikacij enak

$$\left(\partial_f(\partial_g^2 g)\right)e^{h\partial_g}\exp(\partial_f e^{h\partial_g})(g) + (\partial_f^2(\partial_g g)^2)e^{2h\partial_g}\exp(\partial_f e^{h\partial_g})(g). \tag{5.22}$$

Operator je venomer prestavljen v točko ocenitve (5.6) $\mathbf{v} \in \mathcal{V}$, zatorej nas zanima le obnašanje v limiti $h \to 0$. Ko v izrazu (5.22) vzamemo limito, pridemo do operatorja

$$\left(\partial_f(\partial_g^2 g) + \partial_f^2(\partial_g g)^2\right)\exp(\partial_f) : \mathcal{P} \to \partial^2\mathcal{P}(g) \tag{5.23}$$

Tako smo brez vsiljevanja dodatnih pravil izračunali operator drugega odvoda kompozicije z g, direktno na operatorju samem. Ta operator (5.23) pa je sedaj pripravljen za uporabo na konkretnem programu.

Kot je razvidno iz primera, je računanje z operatorji precej enostavnejše od direktne manipulacije tenzorskih vrst. To nam omogoča enostavno implementacijo nad poljubnimi programskimi prostori. Odvod kompozicije programov lahko izrazimo zgolj z uporabo operatorjev, pravila produkta (5.12) in z odvodom operatorja premika (5.21). Torej eksplicitno poznavanje pravil za odvajanje kompozicij ni potrebno, saj so zakodirana kar v strukturi operatorja kompozicij $exp(\partial_f e^{h\partial_g})$, ki pa ga odvajamo po standardnih pravilih.

Podobno lahko poračunamo tudi višje odvode kompozicij kar na operatorju samem

$$\partial^n(f \circ g) = \left(\frac{d}{dh}\right)^n \exp\left(\partial_f e^{h\partial_g}\right)(g, f)\bigg|_{h=0}. \tag{5.24}$$

5.4 Avtomatsko odvedljivi odvodi

Zmožnost uporabe k-tega odvoda programa $P_1 \in \mathcal{P}$, kot del odvedljivega programa $P_2 \in \mathcal{P}$, se kaže kot uporabna v več znanostih [13]. A zato bi morali biti sposobni obravnavati sam (k-ti) odvod programa $P \in \mathcal{P}$ kot odvedljiv progam $P'^k \in \mathcal{P}$. Prav to pa motivira sledeč izrek.

Izrek 5.4 (Redovna redukcija) *Obstaja takšna preslikava redukcije reda $\phi : \mathcal{P}_n \to \mathcal{P}_{n-1}$, da sledeč diagram komutira*

$$\begin{tikzcd} \mathcal{P}_n \arrow[r, "\phi"] \arrow[d, "\partial"] & \mathcal{P}_{n-1} \arrow[d, "\partial"] \\ \mathcal{P}_{n+1} \arrow[r, "\phi"] & \mathcal{P}_n \end{tikzcd} \tag{5.25}$$

in zadošča enakosti

$$\forall_{P_1 \in \mathcal{P}_0} \exists_{P_2 \in \mathcal{P}_0} \Big(\phi^k \circ e_n^\partial(P_1) = e_{n-k}^\partial(P_2)\Big)$$

za vsak $n \geq 1$, kjer je e_n^∂ projekcija operatorja e^∂ na nabor $\{\partial^n\}$.

Posledica 5.2 (Odvedljiv odvod) *Po izreku 5.4, je n-krat odvedljivi k-ti odvod programa $P \in \mathcal{P}_0$ moč izrazit kot*

$$^nP^{k\prime} = \phi^k \circ e_{n+k}^\partial(P) \in \mathcal{P}_n$$

Torej smo pridobili sposobnost zapisovanja odvedljivih programov, ki operirajo na odvedljivih odvodih drugih programov. To je ključna sposobnost, na pomembnost katere (in pomanjkanje v večini modelov) opozarjajo drugi avtorji [26, 25].

5.5 Funkcijske transformacije in interpreterji programov

V pričujočem razdelku bomo pokazali, kako lahko z izpeljano teorijo razvijemo nove pristope, v katerih nato težje izračune prepustimo učnim algoritmom in tako pridemo do novih modelov globokega učenja. Idejo bomo predstavili na primeru izpeljave *nevronskih programskih interpreterjev* [27].

Predpostavimo strojno opremo H, ki je optimizirana za nabor funkcij $F = \{f_i : \mathcal{V} \to \mathcal{V}\}$. Nabor F je specificiran s strani proizvajalca.

S tehnološkimi napredki je menjava strojne opreme pogost pojav, ki lahko vodi v upad učinkovitosti. Zatorej bi želeli transformacijo programa $P \in \mathcal{P}$ v bazi F. Pogosto (predvsem v strojnem in globokem učenju) se zadovoljimo s sub-optimalnim algoritmom, ki je učinkovit na strojni opremi H. Sub-optimalnost algoritma je odvisna od nabora F, od tega, ali napenja $\mathcal{P}$ ali ne. Klasični primer transformacije baze je Fourierjeva transformacija.

Z orodji, ki smo jih razvili v tem delu je problem rešljiv preko linearne algebre. Naj e_n^∂ označuje projekcijo operatorja e^∂ na prvih n baznih vektorjih $\{\partial^i\}$. Po Izreku 5.2 lahko konstruiramo preslikavo (5.7) iz prostora programov v prostor polinomov z neznankami v $\mathcal{V}^k$. Naj $\mathcal{X} = \{p_i\}$ označuje bazo prostora polinomov $\mathcal{V} \to \mathcal{V}$.[3] Operator $e_n^\partial(P \in \mathcal{P})$ lahko interpretiramo kot vektor linearnih kombinacij $\mathcal{X}$.

Definirajmo tenzor $T_{\mathcal{X}F}$ transformacije baze $F \to \mathcal{X}$ s

$$T_{\mathcal{X}F} = p_1 \otimes e_n^\partial(f_1)^* + p_2 \otimes e_n^\partial(f_2)^* + \cdots + p_n \otimes e_n^\partial(f_n)^*. \tag{5.26}$$

Torej je tenzor transformacije baze $\mathcal{X} \to F$ njegov inverz

$$T_{F\mathcal{X}} = T_{\mathcal{X}F}^{-1}. \tag{5.27}$$

Za specifični nabor F moramo tenzor (5.27) poračunati le enkrat in ga lahko potem uporabljamo za transformacije poljubnih programov. Koordi-

[3] Ena od izbir bi bila baza monomov, sestavljena iz elementov $\mathbf{e}_i \otimes \prod_{\alpha,\forall_j} x_{\alpha_j}$, kjer $\mathbf{e}_i$ napenjajo $\mathcal{V}$, x_i napenjajo $\mathcal{V}^*$ in je α multi-indeks.

nate programa $P \in \mathcal{P}$ v bazi F so

$$P_F = T_{F\mathcal{X}} \cdot e^{\partial}(P). \tag{5.28}$$

Izraz (5.28) predstavlja koordinate programa P v bazi F. Torej lahko program izrazimo kot linearno kombinacijo f_i, s komponentami P_F za koeficiente.

$$P = \sum_{i=0}^{n} P_{Fi} f_i \tag{5.29}$$

Če F ne napenja $\mathcal{P}$, ali pa smo uporabili projekcijo operatorja $e^{\partial}_{n<N}$, je izraz (5.29) kljub temu najboljši možni približek programa, na komponentah $\{\partial^n\}$, v bazi F.

Na izpeljani postopek lahko gledamo kot na prevajanje programov iz enega nabora funkcij na drugega. Izračun tenzorjev transformacije (5.27) baze pa lahko prepustimo učnim algoritmom, kot funkcijo vhoda programa. Tako pridemo do različice *nevronskih programskih interpreterjev* [27], kot smo napovedali v začetku razdelka.

5.6 Kontrolne strukture

Čeprav je v našem modelu mogoče konstruirati iteratorje s predstavljenimi operatorji, bomo v tem razdelku predstavili, kako se naša teorija vpleta v kontrolne strukture imperativnih jezikov. Primer uporabe takšne vpletenosti je implementacija v *C++*, ki je predstavljena v razdelku 4.4.

Kontrolne strukture direktno ne spreminjajo vrednosti spremenljivk, ampak spreminjajo izvršilno drevo programa. Zato jih interpretiramo kot definicije (odsekoma zveznih) zlepkov. Vsaka kontrolna struktura deli prostor parametrov na različne domene, znotraj katerih je izvajanje programa venomer enako. Celoten program pa deli prostor vseh mogočih parametrov na končni nabor domen $\{\Omega_i; \quad i = 1, \dots k\}$. Zato je v splošnem program lahko definiran kot

$$P(\vec{x}) = \begin{cases} P_{n_1 1} \circ P_{(n_1-1)1} \circ \ldots P_{11}(\vec{x}); & \vec{x} \in \Omega_1 \\ P_{n_2 2} \circ P_{(n_2-1)2} \circ \ldots P_{12}(\vec{x}); & \vec{x} \in \Omega_2 \\ \vdots & \vdots \\ P_{n_k k} \circ P_{(n_k-1)k} \circ \ldots P_{1k}(\vec{x}); & \vec{x} \in \Omega_k \end{cases} \tag{5.30}$$

Operator e^∂ (v neki točki) programa P je seveda odvisen od začetnih parametrov $\vec{x}$ in je izrazljiv kot zlepek znotraj domen Ω_i

$$e^\partial P(\vec{x}) = \begin{cases} e^\partial P_{n_1 1} \circ e^\partial P_{(n_1-1)1} \circ \ldots \circ e^\partial P_{11}(\vec{x}); & \vec{x} \in \mathrm{int}(\Omega_1) \\ e^\partial P_{n_2 2} \circ e^\partial P_{(n_2-1)2} \circ \ldots \circ e^\partial P_{12}(\vec{x}); & \vec{x} \in \mathrm{int}(\Omega_2) \\ \vdots & \vdots \\ e^\partial P_{n_k k} \circ e^\partial P_{(n_k-1)k} \circ \ldots \circ e^\partial P_{1k}(\vec{x}); & \vec{x} \in \mathrm{int}(\Omega_k) \end{cases} \tag{5.31}$$

Izrek 5.5 *Vsak program $P \in \mathcal{P}$, ki vsebuje kontrolne strukture, je neskončnokrat odvedljiv na domeni $\Omega = \bigcup\limits_{\forall i} \mathrm{int}(\Omega_i)$.*

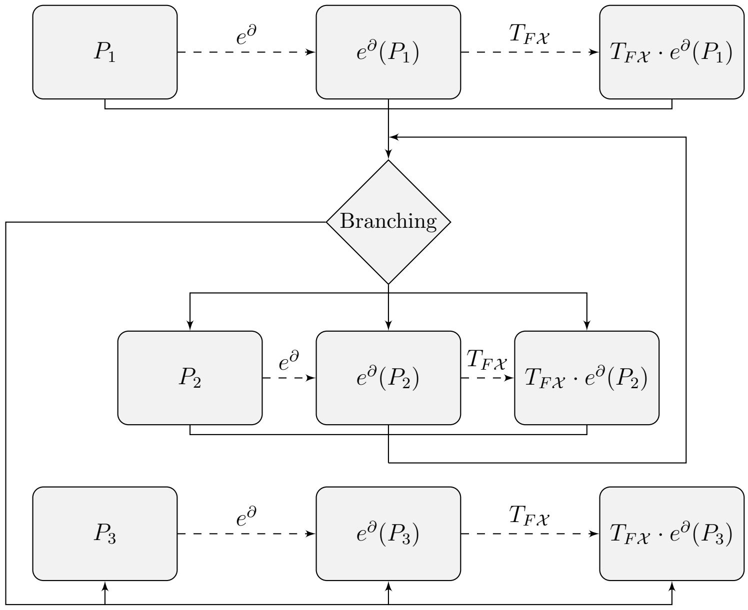
Dokaz. Notranjost vsake domene Ω_i je odprta. Ker je celotna domena $\Omega = \bigcup\limits_{\forall i} \mathrm{int}(\Omega_i)$ unija odprtih množic, je posledično odprta tudi sama. Zatorej so vse evaluacije izvršene v eni od odprtih množic, kar efektivno odstrani robove, kjer bi lahko naleteli na težave. Izrek sledi direktno iz dokaza Izreka 4.1 skozi argument (4.11). □

Vejitev v domeni programa (5.30) se izvede s pogojnimi izjavami. Vsaka pogojna izjava povzroči vejitev v izvršilnem drevesu programa.

Trditev 5.2 *Kardinalnost množice domen $\Omega = \{\Omega_i\}$ je enaka $|\{\Omega_i\}| = 2^k$, kjer je k število vejitvenih točk v programu.*

Predmet pouka tega razdelka je uporaba izpeljanih teoremov za linearno obravnavanje vejitev, da se izognemo eksponentni grožnji Trditve 5.2.

Izrek 5.6 *Program $P \in \mathcal{P}$ je lahko ekvivalentno predstavljen z $2n+1$ aplikacijami operatorja e^∂, na $2n+1$ analitičnih programov, kjer je n število vejitev znotraj programa.*



Slika 5.1: Diagram transformacij

Dokaz. Izvorna koda programa $P \in \mathcal{P}$ je predstavljiva z usmerjenim grafom, kot je razvidno na Sliki 5.1. Vsaka vejitev povzroči delitev v izvršitvenem drevesu, kamor se program po izvedbi veje tudi vrne. Po Izreku 5.3 je vsaka od teh vej lahko videna kot program p_i, za katerega velja

$$e^\partial(p_n \circ p_{n-1} \circ \cdots \circ p_1) = e^\partial(p_n) \circ e^\partial(p_{n-1}) \circ \cdots \circ e^\partial(p_1)$$

po Izreku 5.3.

Zatorej izvorna koda vsebuje $2n$ odvedljivih vej, od prve vejitve dalje, na katere je potrebno aplicirat operator e^∂. Z upoštevanjem prve delitve je to skupaj $2n + 1$. Po Izreku 4.1, je vsaka od teh vej analitična. $\square$

Sliki operatorja e^∂ in $T_{F\mathcal{X}}$ sta elementa prvotnega programskega prostora $\mathcal{P}$, in sta lahko komponirani. Zatorej je za $P = p_3 \circ p_2 \circ p_1$, slednji izraz smiseln

$$P = \Big(p_3 \circ e^\partial(p_2) \circ T_{F\mathcal{X}} e^\partial(p_1)\Big) \in \mathcal{P}.$$

Enako velja za vse permutacije uporab operatorjev e^∂, $T_{F\mathcal{X}}$ in id, kot je razvidno v diagramu Slike 5.1.

Opomba 5.6 *V praksi vedno uporabljamo projekcije operatorja e^∂ na neki končni red n, tj. e^∂_n. Zato moramo biti pazljivi na veljavnost sledeče relacije*

$$e^\partial_m(P_2) \circ e^\partial_n(P_1) = e^\partial_k(P_2 \circ P_1) \iff 0 \leq k \leq \min(m, n)$$

ko komponiramo dve sliki operatorjev, ki sta bila projicirana na različna podprostora.

6

Sklep in nadaljnje delo

Navdahnjeni s strani podvigov Feynmana [10] in Heavisidea [7] v fiziki pred nami, smo uporabili operatorski račun na programskih prostorih. S tem smo omogočili sklepanje o analitičnih lastnostih programov preko zgolj algebraičnih prijemov, kar je olajšalo postopek implementacije. Naš doprinos je bil operator komponiranja programov, ki posplošuje tako vnaprejšnji, kot vzvratni način avtomatskega odvajanja poljubnega reda z enim samim operatorjem. Nato smo uporabo izpeljane algebre in operatorskega računa predstavili tako, da smo izračune ter manipulacije izvedli na operatorjih samih, preden je bil operator apliciran na konkretni program.

Algebraični jezik, ki je predstavljen v našem delu, nadomesti kompleksne pojme s preprostimi izrazi in omogoči formulacijo ustreznih, smiselnih algebraičnih enačb.

Po takšnem postopku smo izpeljali funkcijske transformacije programov v poljubni bazi. Na primer, postopek se izkaže za koristno orodje, ko programje prilagajamo na specifično strojno opremo. Vse takšne formulacije so *invariante* na izbiro programskega prostora in na trenutno točko v izvajanju programa. S tem smo vpeljali pojem splošne kovariance v programiranje. To vpeljavo smo uporabili pri svojem snovanju metod, ki omogočajo medsebojno prepletenost uporabljenih trasformacij. Takšne metode omogočajo prehajanje med transformiranimi oblikami in prvotnim programom.

6.1 Snovanje novih modelov in arhitektur

Algebraični jezik, ki smo ga razvili v tem delu, omogoča uporabo ustaljenih prijemov matematične analize na programju. Po tem, ko je metoda izpeljana, pa lahko računsko zahtevnejše dele postopka nadomestimo z učnim algoritmom, ki na podlagi vhoda vrača približno rešitev. Tako smo postopali v razdelku 5.5, kjer smo s svojo algebro izpeljali menjavo baze programa. Ob zamenjavi izračuna tenzorja transformacije baze z učnim algoritmom, smo izpeljali postopek, ki je ekvivalenten *nevronskim interpreterjem programov* [27], le da imamo na svoj način tudi teoretično utemeljitev postopka. Podobno lahko postopamo tudi pri drugih matematičnih prijemih in tako izpeljemo nove arhitekture ter modele globokega učenja.

6.2 Druga teoretična poizvedovanja

Z algebraičnim jezikom, ki smo ga razvili v svojem delu, lahko izpeljemo *nevronske tenzorske vrste* [35], ki so v tem delu izpuščene, a jih bralec lahko poišče v avtorjevem članku [35]. Z njimi je dosežena povezava med programi in tenzorskimi vrstami: vsaka tenzorska mreža je nevronska tenzorska vrsta nekega programa. Prav to pa zapira vrzel med matematično analizo in programiranjem, ki smo jo omenili v uvodu. Hkrati pa nudi nov način izražanja nevronskih modelov, ki omogoča teoretični vpogled v formulirane arhitekture globokega učenja.

Literatura

[1] A. Abdelfattah et al. High-performance tensor contractions for gpus. *Procedia Computer Science*, 80:108 – 118, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[2] Ralph Abraham, Jerrold E. Marsden, and Tudor Ratiu. *Manifolds, Tensor Analysis, and Applications (Applied Mathematical Sciences) (v. 75)*. Springer, 1988.

[3] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[4] Luca Bertinetto, Jack Valmadre, João F. Henriques, Andrea Vedaldi, and Philip H. S. Torr. Fully-convolutional siamese networks for object tracking. *arXiv:1606.09549*, 2016.

[5] Andreas Blass. Seven trees in one. *arXiv:math/9405205*, 1994.

[6] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *SIGNLL Conference on Computational Natural Language Learning (CONLL), 2016*, 2015.

[7] John R. Carson. The heaviside operational calculus. *Bell System Technical Journal*, 1(2):43–55, 1922.

[8] Atilim Gunes Baydin et. al. Automatic differentiation in machine learning: a survey. *arXiv:1502.05767*, 2015.

[9] Martín Abadi et.al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

[10] Richard P. Feynman. An operator calculus having applications in quantum electrodynamics. *Phys. Rev.*, 84:108–128, Oct 1951.

[11] Marcelo Fiore and Tom Leinster. Objects of categories as complex numbers. *Advances in Mathematics 190 (2005), 264-277*, 2002.

[12] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[13] Mark Girolami and Ben Calderhead. Riemann manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society*, 2011.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[15] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv:1410.5401*, 2014.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv:1512.03385*, 2015.

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

[18] Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, July 2014.

[19] André Joyal. Une théorie combinatoire des séries formelles. *Advances in mathematics*, 42(1):1–82, 1981.

[20] Kamil A. Khan and Paul I. Barton. A vector forward mode of automatic differentiation for generalized derivative evaluation. *Optimization Methods and Software*, 30(6):1185–1212, 2015.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[22] Henry W. Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *arXiv:1608.08225*, 2016.

[23] O'Hanian, Hans C., Ruffini, and Remo. *Gravitation and Spacetime (2nd ed.)*. W. W. Norton, 1994.

[24] Ankur P. Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. *arXiv:1606.01933*, 2016.

[25] Barak A. Pearlmutter and Jeffrey M Siskind. Putting the Automatic Back into AD: Part I, What's Wrong (CVS: 1.1). *ECE Technical Reports.*, 2008.

[26] Barak A. Pearlmutter and Jeffrey M Siskind. Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast (CVS: 1.1). *ECE Technical Reports.*, May 2008.

[27] Scott Reed and Nando de Freitas. Neural programmer-interpreters. *International Conference on Learning Representations (ICLR)*, 2015.

[28] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *arXiv:1506.01497*, 2015.

[29] J. L. Synge and A. Schild. *Tensor Calculus (Dover Books on Mathematics)*. Dover Publications, 2012.

[30] Chris Taylor. Algebra of algebraic data types, 2013. dosegljivo na http://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types/.

[31] Norbert Wiener. The operational calculus. *Mathematische Annalen*, 95:557–584, 1926.

[32] Žiga Sajovic. dcpp, 2016. https://github.com/zigasajovic/dCpp.

[33] Žiga Sajovic. dcpp: Infinite differentiability of conditionals, recursion and all things c++, 2016. https://zigasajovic.github.io/dCpp/.

[34] Žiga Sajovic. Implementing operational calculus on programming spaces. *arXiv e-prints*, arXiv:1612.0273, 2016.

[35] Žiga Sajovic and Martin Vuk. Operational calculus on programming spaces. arXiv:1610.07690. Poslano v Journal of Machine Learning Research.