UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Računalništvo in matematika – 2. stopnja

Sebastian Mežnar

UČINKOVIT GENERATIVNI MODEL ZA ALGEBRAJSKE IZRAZE IN ODKRIVANJE ENAČB

Magistrsko delo

Mentor: prof. dr. Ljupčo Todorovski

Ljubljana, 2022

Zahvala

Zahvaljujem se mentorju, prof. dr. Ljupčotu Todorovskemu, za napotke, ideje in popravke, ki mi jih je dal med pisanjem magistrske naloge.

Rad bi se zahvalil tudi dekletu, družini, članom skupine ED iz Odseka za tehnologije znanja (E8) in prof. dr. Sašu Džeroskemu za podporo ter pomoč.

Kazalo

Program dela

Pogosti pristop k odkrivanju enačb je dvofazni „ustvari in preizkusi“. Faza ustvarjanja je zasnovana na generatorju aritmetičnih izrazov, ki se pojavijo na desni strani enačbe za podano ciljno spremenljivko. Faza preizkušanja določi neznane vrednosti konstantnih parametrov aritmetičnega izraza tako, da se vrednosti ciljne spremenljivke izračunane z aritmetičnim izrazom čim bolj ujemajo z opazovanimi. Generatorji aritmetičnih izrazov so pogosto del algoritma za odkrivanje enačb ali pa slonijo na kontekstno-neodvisnih gramatikah, ki določajo prostor možnih aritmetičnih izrazov [6]. Cilj magistrske naloge je razvoj generativnega modela, ki bi slonel na globokih variacijskih samokodirnikih [25] za ustvarjanje aritmetičnih izrazov. Razviti generativni model bi preizkusili na standardnih problemih odkrivanja enačb iz podatkov.

Osnovna literatura

[6] J. Brence, L. Todorovski in S. Džeroski, *Probabilistic grammars for equation discovery*, Knowledge-Based Systems **224** (2021) 107077

[25] D. P. Kingma in M. Welling, *An introduction to variational autoencoders*, Foundations and Trends® in Machine Learning **12**(4) (2019) 307–392, doi: 10.1561/2200000056

Podpis mentorja:

Učinkovit generativni model za algebrajske izraze in odkrivanje enačb

Povzetek

Odkrivanje enačb se ukvarja z iskanjem algebrajskih izrazov, ki se prilegajo danim podatkom. V nalogah odkrivanja enačb damo pogosto velik poudarek na generiranje izrazov. Čeprav so se izrazi v preteklosti generirali predvsem z kontekstno neodvisnimi gramatikami, evolucijskimi algoritmi in ostalimi pristopi, pa nedavno v ospredje prihajajo globoki generativni modeli. Prvi poskusi generiranja diskretnih, strukturiranih podatkov z globokimi generativnimi modeli vključujejo variacijske samokodirnike (CVAE) za preproste, neomejene nize simbolov in variacijske samokodirnike gramatik (GVAE), ki z uporabo kontekstno neodvisnih gramatik izhod dekodirnika sintaktično omejijo. V magistrskem delu predstavimo variacijski samokodirnik hierarhij (HVAE), ki v nasprotju s prejšnjimi pristopi izhod dekodirnika omeji z binarnimi izraznimi drevesi. Drevesa zakodiramo in dekodiramo s prilagojenima različicama rekurentne nevronske mreže z vrati. Trdimo, da lahko pristop HVAE naučimo bolj učinkovito kot pristopa CVAE in GVAE. To trditev potrdimo z empiričnim vrednotenjem, kjer je HVAE pri rekonstrukciji bolj uspešen kot druga pristopa kljub manjši učni množici in nižji dimenziji latentnega vektorja. Slednje simbolni regresiji dovoljuje bolj učinkovito uporabo Bayesove optimizacije za odkrivanje kompleksnih enačb iz podatkov.

Efficient generative model for algebraic expressions and equation discovery

Abstract

Equation discovery searches for algebraic expressions that model the given data. In equation discovery tasks, strong emphasis is usually put on the generation of expressions. Historically, expressions are generated by using context-free grammars, evolutionary algorithms and other approaches, but recently generators based on deep learning started to emerge. First attempts at generating discrete, structured data with deep generative models include variational autoencoders (VAE) for simple, unconstrained character sequences, and grammar VAEs, which employ context-free grammars to syntactically constrain the output of the decoder. In contrast, the hierarchical VAE (HVAE) proposed in this paper constrains the output of the decoder to binary expression trees. These trees are encoded and decoded with two simple extensions of gated recursive units. We conjecture that the HVAE can be trained more efficiently than sequential and grammar based VAEs. Indeed, the experimental evaluation results show that the HVAE can be trained with less data and in a lower-dimensional latent space, while still significantly outperforming other approaches. The latter allows for efficient symbolic regression via Bayesian optimization in the latent space and the discovery of complex equations from data.

Math. Subj. Class. (2010): 68T05, 68T37, 62F15

Ključne besede: odkrivanje enačb, simbolna regresija, nevronske mreže, generativni modeli, variacijski samokodirniki, strojno učenje, globoko učenje, Bayesova optimizacija

Keywords: equation discovery, symbolic regression, neural networks, variational autoencoders, generative models, machine learning, deep learning, Bayesian optimization

1 Uvod

Znanstveniki in inženirji se pogosto ukvarjajo z iskanjem matematičnih modelov, ki opišejo obnašanje opazovanih pojavov. Kljub nedavnemu napredku strojnega učenja v smeri zelo točnih modelov črnih škatel, ki ne ponudijo poglobljenega razumevanja opazovanega pojava, je iskanje preprostih matematičnih modelov še vedno relevantno, saj le-te lahko poglobijo naše znanje in razumevanje sveta. Poleg tega lahko z enačbo, ki pravilno modelira pojav, zagotovimo točnost napovedi na celotni domeni problema tudi izvan območja opazovanj, ki smo jih uporabili za učenje. Zgodovina znanosti pokaže, da je iskanje omenjenih modelov zapletena naloga, ki poleg tega, da vzame veliko časa, zahteva tudi veliko izkušenj, intuicijo in znanje s področja opazovanega pojava. V okviru strojnega učenja se je zato razvilo področje odkrivanja enačb (angl. *equation discovery*) znano tudi kot simbolna regresija (angl. *symbolic regression*), ki se ukvarja z iskanjem enostavnih enačb iz opazovanih podatkov.

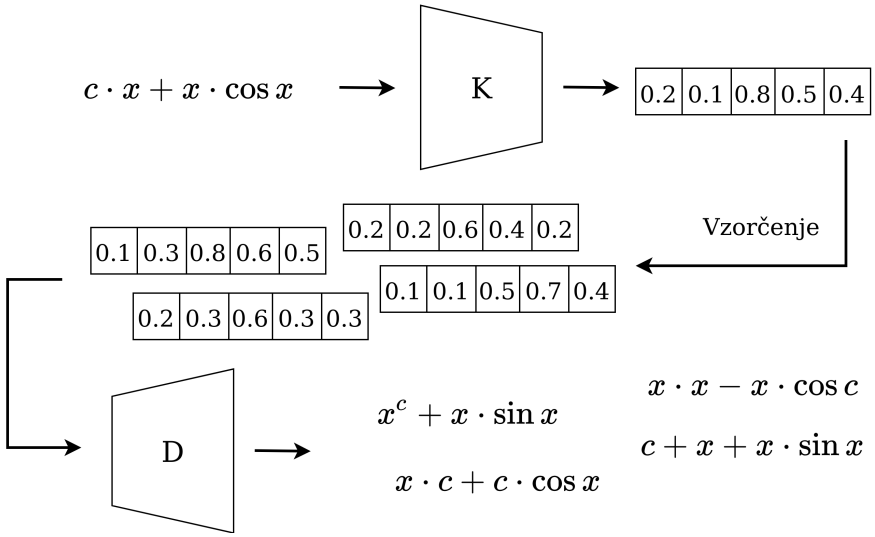
Iskanje enačb, ki se podatkom prilegajo je računsko zahteven problem, saj je prostor možnih izrazov zelo velik in neurejen. Algoritmi za odkrivanje enačb zato pogosto generirajo veliko množico izrazov in izberejo tiste, ki se dobro prilegajo podanim podatkom. V magistrskem delu se osredotočimo predvsem na generiranje izrazov. Težava trenutnih pristopov je, da pogosto izraze generirajo naključno ali pa po pristopu grobe sile oziroma izčrpnega naštevanja. Taki pristopi le redko uporabljajo povratne informacije o prileganju že ovrednotenih izrazov in je zato generiranje izrazov neusmerjeno: obetavni izrazi, ki se dobro prilegajo podatkom so enakopravni tistim, ki se ne prilegajo podatkom. Posledično tako pristopi posvetijo veliko časa preiskovanju prostora možnih izrazov.

Pristop k odkrivanju enačb razvit v okviru magistrske naloge se bo pri iskanju osredotočal na dele prostora, ki vsebujejo obetavne izraze z dobrim prileganjem podanim podatkom. Slonel bo na modelih za vpetje (angl. *embedding*) algebrajskih izrazov v nizko-dimenzionalen evklidski prostor. Evklidski prostor lahko nato preiskujemo z metodami kot je Bayesova optimizacija, ki pri preiskovanju upošteva informacijo o že ovrednotenih izrazih in iskanje usmerja v obetavne dele preiskovalnega prostora. Dva modela za vpetje strukturiranih izrazov, ki sledita tej ideji, sta CVAE [19] in GVAE [29]. CVAE v evklidski prostor vpne nize simbolov, GVAE pa zaporedja prepisovalnih pravil kontekstno neodvisne gramatike (angl. *context-free grammars*). CVAE pri generiranju ne zagotovi sintaktično pravilnost izrazov, GVAE pa zagotovi le pravilnost izrazov, ki jih z gramatiko lahko izpeljemo v omejenem številu prepisovalnih korakov.

V magistrski nalogi predstavimo generativni model, ki zagotovi sintaktično pravilnost generiranih izrazov z uporabo dvojiških izraznih dreves. Ta kodiramo in dekodiramo z enostavno prilagoditvijo rekurentne nevronske mreže z vrati (angl. *gated recurrent unit, GRU*) [10]. Naš variacijski samokodirnik HVAE (angl. *hierarchical variational autoencoder*) doseže nižjo rekonstrukcijsko napako od pristopov CVAE in GVAE, kljub manjši učni množici in nižji dimenziji latentnega prostora. Slednja lastnost HVAE je zelo pomembna, saj Bayesova optimizacija bolje deluje v prostorih z nižjo dimenzijo.

Shema na sliki 1 prikazuje delovanje HVAE za generiranje izrazov, ki so sintaktično podobni podanemu začetnemu izrazu. Izraz najprej zakodiramo s kodirnikom.

Tako dobimo vektorsko predstavitev izraza, ki jo uporabimo kot aritmetično sredino Gaussove porazdelitve iz katere vzorčimo nove vektorje. Ko te pošljemo čez dekodirnik dobimo izraze, ki so sintaktično podobni začetnemu.



Slika 1: Shema delovanja samokodirnika HVAE za generiranje algebrajskih izrazov, ki so podobni podanemu izrazu. Trapezoid označen s črko K predstavlja kodirnik, trapezoid s črko D pa dekodirnik.

Prispevki magistrskega dela so:

- Zasnujemo in implementiramo variacijski samokodirnik HVAE za generiranje algebraičnih izrazov in, bolj splošno, hierarhičnih struktur z globokimi nevronskimi mrežami oz. prilagoditvijo rekurentne nevronske mreže z vrati;
- Implementiramo algoritem za odkrivanje enačb z Bayesovsko optimizacijo za preiskovanje latentnega evklidskega prostora v katerega samokodirnik HVAE vpne algebrajske izraze;
- Empirično ovrednotimo delovanje razvitih pristopov na nalogah rekonstrukcije algebrajskih izrazov in odkrivanja enačb.

Magistrska naloga je sestavljena iz petih poglavij. V 2. poglavju naredimo pregled del iz odkrivanja enačb in generativnih modelov ter opišemo matematično ozadje nevronskih mrež, variacijskih samokodirnikov in Bayesove optimizacije. V 3. poglavju predstavimo naš pristop ter podrobnosti, ki se navezujejo na predstavitev izraza, učenje generativnega modela, generiranje izrazov in odkrivanje enačb z Bayesovo optimizacijo. V 4. poglavju najprej predstavimo okvir naše empirične evalvacije, nato pa rezultate, ki pokažejo kvaliteto rekonstrukcije, gladek prehod med izrazi in primere rekonstrukcije znanih enačb iz podatkov. Na koncu v 5. poglavju povzamemo delo in podamo ideje za nadaljnje delo.

2 Ozadje

V tem poglavju najprej predstavimo področje odkrivanja enačb in kako naš pristop umestimo vanj. Nato predstavimo globoke generativne modele, variacijske samo-

kodirnike in dosedanje pristope za vpetje in generiranje strukturiranih podatkov. Nazadnje predstavimo še Bayesovo optimizacijo, ki jo v delu uporabimo za preiskovanje latentnega prostora.

2.1 Odkrivanje enačb

Odkrivanje enačb oziroma simbolna regresija (angl. *symbolic regression*) je področje strojnega učenja (angl. *machine learning*), ki se ukvarja z iskanjem algebrajskih izrazov, ki se prilegajo podanim podatkom. Ti podatki so numerični in ponavadi predstavljajo meritve opazovanega pojava. Nadaljnja predpostavka je, da se meritve nanašajo na podani nabor spremenljivk $x_1, x_2, \ldots x_m$. V najpreprostejši različici problema, tisti na katero se bomo osredotočili, želimo poiskati enačbo oblike

$$x_i = f(x_1, ..., x_{i-1}, x_{i+1}, ..., x_m), \tag{2.1}$$

s katero lahko izračunamo vrednost *ciljne spremenljivke* x_i iz vrednosti preostalih spremenljivk x_j, $j \neq i$. Želimo, da se vrednosti x_i, ki so izračunane z uporabo zgornje enačbe, dobro prilegajo izmerjenim podatkom za x_i. Hkrati si želimo, da bi bil algebrajski izraz na desni strani enačbe čim bolj preprost (kratek). Preprosti izrazi so namreč bolj interpretabilni in pogosto dosežejo nižjo napako na novih, izmerjenih podatkih, ki niso bili na voljo pri učenju enačbe.

Algoritmi za odkrivanje enačb pogosto sledijo splošnemu pristopu *ustvari in preizkusi* (angl. *generate-and-test*). Korak *ustvari* tvori strukture kandidatnih algebrajskih izrazov, t.j. izrazov, v katerih vrednosti konstantnih členov še niso določene (na primer $c \cdot x + \sin(c \cdot x)$, kjer je c znak za konstanto). Korak *preizkusi* nato poišče optimalne vrednosti konstant in s tem poskrbi, da je razlika med vrednostmi izračunanega izraza in vrednostmi ciljne spremenljivke čim manjša. Problem dodatno oteži dejstvo, da težko določimo podobnost med dvema izrazoma. Že s spremembo konstant v koraku *preizkusi* se lahko namreč vrednost izraza drastično spremeni.

V našem delu se osredotočimo predvsem na korak *ustvari*, saj ta bolj vpliva na hitrost iskanja in kvaliteto dobljene enačbe. V grobem lahko metode za odkrivanje enačb ločimo glede na tri lastnosti: predstavitev algebrajskih izrazov, način preiskovanja prostora možnih izrazov in integracija domenskega znanja.

Predstavitev izraza določi kako bo algoritem za odkrivanje enačb sestavljal algebrajske izraze in omejil prostor možnih izrazov. Dober primer omejitve prostora je algoritem SINDy [9], ki s transformacijami podanih spremenljivk tvori nove spremenljivke in nad podanimi in novimi spremenljivkami uporabi redko linearno regresijo (angl. *sparse linear regression*). Oblika dobljenega algebrajskega izraza na desni strani enačbe je tako omejena na linearne kombinacije novih in transformiranih spremenljivk. Podobno omejitev uporablja LAGRANGE [13], ki lahko s transformacijami vpelje tudi časovne odvode podanih spremenljivk in tako omogoča odkrivanje diferencialnih enačb. Algoritem E* [35] se omeji na končno množico vnaprej podanih struktur izrazov, ki se pogosto pojavljajo v znanstveni literaturi. COPER [26] in SDS [47] pa uporabljata omejitve, ki slonijo na pravilni kombinaciji merskih enot opazovanih spremenljivk.

Pri vseh algoritmih iz prejšnjega odstavka je predstavitev izraza prilagojena omejitvam prostora možnih izrazov. Druga skupina algoritmov za simbolno regresijo izraze predstavi z izraznimi drevesi (glej poglavje 3.1). To predstavitev sta

popularizirala Schmidt in Lipson [36]. Njun algoritem Eurequa temelji na mutiranju in križanju izraznih dreves in izboljšanju populacije le-teh z genetskim algoritmom [27]. Predstavitev z izraznimi drevesi uporablja tudi algoritem Bayesovega znanstvenika [18]. Izrazna drevesa namreč omogočajo predstavitev poljubnega algebrajskega izraza iz podanega nabora spremenljivk, operatorjev in funkcij, a obenem drastično povečajo prostor možnih izrazov.

Naslednja veja algoritmov za predstavitev izraza uporabi kontekstno neodvisne gramatike. Gramatike določajo prepisovalna pravila, s pomočjo katerih lahko izraz izpeljemo iz nabora osnovnih simbolov (operatorjev, funkcij, spremenljivk in konstant). Algoritem LAGRAMGE [43] definira operator izostritve (angl. *refinement operator*) za urejanje prostora možnih izpeljav in z njim posledično uredi tudi prostor izpeljanih algebrajskih izrazov na desni strani odkritih enačb. ProGED [6] uporabi verjetnostne gramatike zato, da definira verjetnostno porazdelitev nad prostorom možnih izpeljav oziroma algebrajskih izrazov. Grammar Variational Autoencoder (GVAE) je generator algebrajskih izrazov, ki vpne izpeljave definirane z gramatiko v latentni vektorski prostor in nato z vzorčenjem v tem prostoru omogoča naključno generiranje algebrajskih izrazov.

Nazadnje, algoritmi pogosto dovoljujejo implementacijo preprostih omejitev, ki dobljene izraze optimizirajo. Primera teh sta omejitvi, da se ob operatorju ne pojavita dve konstanti in da je izraz daljši od štirih in krajši od tridesetih znakov. Te omejitve se lahko upoštevajo med generacijo ali pa po njej.

Kot smo omenili, lahko algoritme za odkrivanje enačb razvrščamo tudi glede na **način preiskovanja prostora izrazov**. Algoritmi, ki slonijo na močni omejitvi prostora možnih izrazov pogosto uporabljajo pristope grobe sile oziroma sistematičnega preiskovanja. Algoritmi kot LAGRANGE [13], E* [35], COPER [26] in SDS [47] tako zaporedoma in sistematično generirajo možne izrazne strukture in jih preizkušajo, dokler ne najdejo tistega, ki se najbolje prilega podatkom. SINDy [9] uporabi redko linearno regresijo, da, v drugem koraku ocenjevanja vrednosti konstant, ugotovi minimalni nabor obstoječih in novih spremenljivk, ki so vključene v enačbo. Preprost pristop je tudi vzorčenje naključnih izrazov (pristop Monte Carlo), ki ga uporablja ProGED [6]. Ta pristop je ob ustrezni omejitvi prostora hiter, njegova implementacija pa enostavna.

Bolj kompleksno je preiskovanje prostora izrazov z genetičnim algoritmom, ki ga pogosto uporabljajo algoritmi za simbolno regresijo, kot je Eureqa [36]. Tak pristop je lahko hiter tudi brez omejitve prostora, a se ob neustreznih parametrih lahko zapelje k lokalnemu minimumu. Algoritem Bayesovega znanstvenika [18] uporabi Bayesov pristop za preiskovanje prostora. Hevristične metode za preiskovanje uporabljata tudi algoritma LAGRAMGE [43], ki sloni na metodi preiskovanja s snopom (angl. *beam search*), in ABACUS [15], ki sloni na podatkovnih hevristikah za tvorjenje novih spremenljivk v algebrajskih izrazih. AI Feynman [44] združuje različne preiskovalne pristope in z njimi učinkovito reši podnaloge, ki jih nato združi v celoto.

Zadnji pristop, ki ga bomo omenili je Bayesova optimizacija [38]. Ta je uporabna predvsem za pristope temelječe na globokem učenju, kot je GVAE [29], ki izraze zakodirajo v latentni vektorski prostor. V njem uporabijo Bayesovo optimizacijo za iskanje minimuma. Ob ustreznem latentnem prostoru je lahko ta pristop hiter in manj občutljiv na lokalne minimume.

Nazadnje, metode razvršamo tudi glede na možnosti in načine **integracije domenskega znanja**. Domensko znanje je pogosto zelo pomembno, saj omeji prostor možnih izrazov in s tem pohitri ter izboljša iskanje. Metode najpogosteje dovoljujejo dodajanje domenskega znanja prek seznama funkcij in simbolov, kot v pristopih, ki slonijo na genetskih algoritmih [48]. Tak pristop dodajanja domenskega znanja ne omeji prostora občutno, a je za uporabnika preprost.

Algoritem ProBMoT temelji na entitetah, ki opišejo sestavne elemente opazovanih dinamičnih sistemov v podani domeni, ter procesih, ki definirajo enačbe za interakcije med entitetami. Formalizem procesnih modelov, ki ga uporablja ProBMoT omogoča predstavitev domenskega znanja za modeliranje dinamičnih sistemov na področjih biologije [14] ali ekologije [7].

LAGRAMGE in ProGED za dodajane domenskega znanja uporabljata gramatike. Te so dovolj splošne za izražanje različnih vidikov domenskega znanja o modeliranju, a je zaradi potrebne matematične/računalniške podlage ta pristop pogosto nedostopen za domenske eksperte iz drugih področij. Še manj je za domenske eksperte dostopen pristop, v katerem je potrebno definirati predhodno distribucijo čez prostor enačb. Ta se pojavi predvsem v algoritmu Bayesovega znanstvenika [18].

Pristop razvit v okviru magistrskega dela izraz predstavi z izraznim drevesom, ki ga nato zakodira v latentni vektorski prostor. Tega preiskujemo z Bayesovo optimizacijo, prostor pa dodatno omejimo s kontekstno neodvisno gramatiko, ki jo uporabimo za generiranje učne množice izrazov.

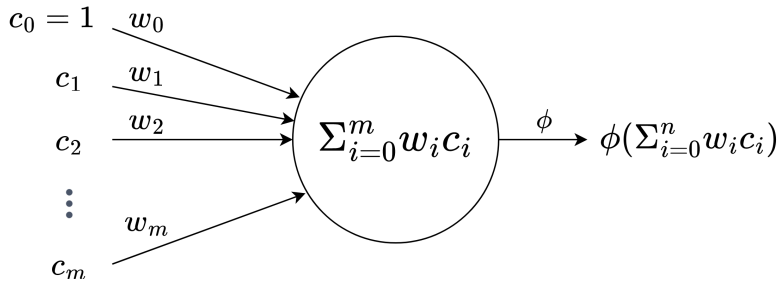
2.2 Generativni modeli

V zadnjih letih so v strojnem učenju vse bolj popularni generativni modeli, ki se iz (običajno velike) množice obstoječih primerov, učijo generirati nove primere, ki niso bili elementi učne množice. Za razliko od klasičnih generativnih modelov, na primer verjetnostnih gramatik, ki jih običajno gradimo ročno, generativni modeli v strojnem učenju temeljijo na globokih nevronskih mrežah, zato jih imenujemo tudi globoki generativni modeli. Ti med drugim vključujejo variacijske samokodirnike (angl. *variational autoencoders*), ki jih bolj podrobno opišemo v nadaljevanju, in generativne nasprotniške mreže (angl. *generative adversarial networks*, GANs). Slednje se naučijo generirati nove primere, ki jih težko razlikujemo od pravih, učnih primerov. Primere uspešne uporabe globokih generativnih modelov med drugim najdemo na področju generiranja besedil [16, 50, 8] in slik [30, 49, 51] ter sinteze govora [20].

Lastnost globokih generativnih modelov je, da večinoma ne potrebujemo označenih učnih primerov. Večina pristopov za učenju generativnih modelov namreč naslavlja nalogo nenadzorovanega ali pa delno nadzorovanega učenja. Samokodirniki se učijo čim bolje rekonstruirati učne podatke, GAN-i pa učne podatke uporabijo le za učenje klasifikacijskega modela, ki ločuje med pravimi (učnimi) in umetnimi (generiranimi) podatki. Torej v tem primeru uporabimo oznako, ki je dodana neoznačenim podatkom zgolj na osnovi tega, kako smo jih pridobili.

2.2.1 Usmerjene in rekurentne nevronske mreže

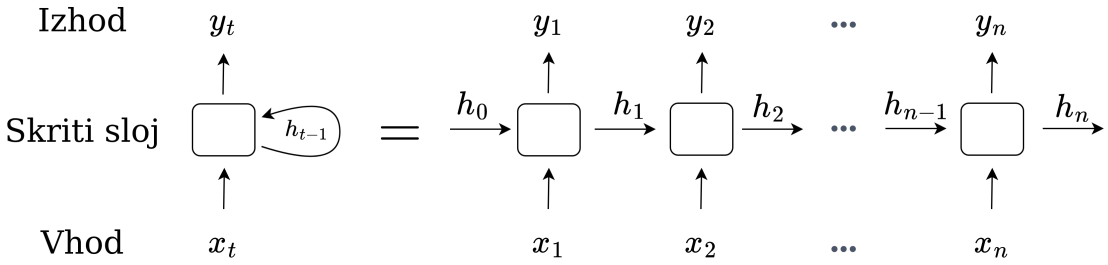
Globoki generativni modeli temeljijo na nevronskih mrežah, ki so sestavljene iz več slojev nevronov. Nevron (prikazan na sliki 2) je osnovni gradnik nevronskih mrež. Vhodi nevrona so numerične vrednosti $c_0, c_1, c_2, \ldots c_m$, ki predstavljajo izhode nevronov iz prejšnjega sloja. Nevron iz vhodov izračuna njihovo utežno vsoto, transformirano z aktivacijsko funkcijo ϕ. Uteži $w_0, w_1, w_2, \ldots w_m$ so pripisane sinapsam, ki povezujejo nevrone in tako določajo strukturo nevronske mreže. Aktivacijska funkcija vpelje nelinearnost v nevronsko mrežo in s tem poveča prostor funkcij, ki jih mreža lahko aproksimira.



Slika 2: Shema nevrona na kateri c_i predstavljajo vhode, w_i uteži povezovalnih sinaps in ϕ aktivacijsko funkcijo. Zmnožek oziroma utež $w_0\, c_0 = w_0$ imenujemo tudi začetna vrednost.

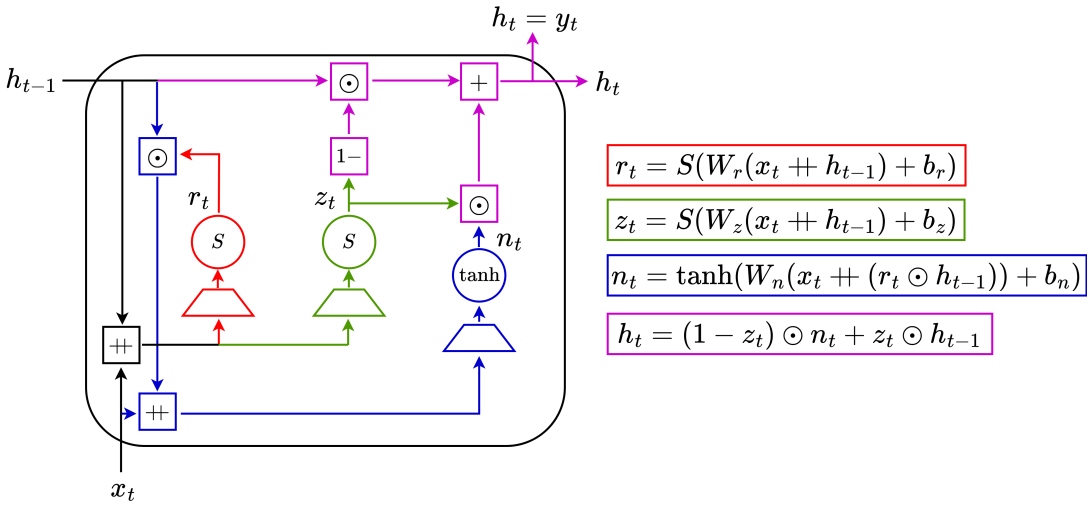
V strojnem učenju pogosto uporabljamo usmerjene nevronske mreže (angl. *feed-forward neural networks*). Gre za mreže s posebno strukturo, kjer so nevroni razvrščeni v zaporedje slojev in nobena dva nevrona iz istega sloja nista medsebojno povezana. Nevroni prvega sloja poskrbijo za zajem vrednosti napovednih spremenljivk učnega primera, zadnji sloj pa za napoved vrednosti ciljne spremenljivke, če gre za mrežo, ki jo uporabljamo kot napovedni model. Sloje med prvim in zadnjim imenujemo skriti sloji. Dva zaporedna sloja sta običajno polno povezana s sinapsami (vsak nevron sloja je povezan z vsakim nevronom predhodnega sloja); polno-povezan sloj iz n nevronov, transformira vhodne m-dimenzionalne podatke iz predhodnega sloja m-tih nevronov v n-dimenzionalne vektorje. Tak sloj lahko prikažemo z matriko $W \in \mathbb{R}^{(m+1)\times n}$, v kateri W_{ij} predstavlja utež za j-to vhodno vrednost i-tega nevrona. Predstavitev uteži z matriko in učinkovita implementacija stohastičnega gradientnega spusta [34] tako omogočata učinkovito učenje vrednosti velikega števila uteži v globokih (večslojnih) nevronskih mrežah.

Slabost polno-povezanega sloja je, da potrebuje vhod in izhod z vnaprej določeno velikostjo. Podatki s spremenljivo dolžino, kot so nizi znakov, video posnetki, ali časovne vrste, so torej lahko uporabljeni le ob *diskretizaciji* ali predhodnem vpetju v prostor z vnaprej določeno dolžino. Zaradi težke prilagoditve, ki pogosto dosega slabe rezultate, se zato za podatke s spremenljivo dolžino uporabijo rekurentne nevronske mreže (angl. *reccurent neural networks*). Naj bo $x_1 x_2 \ldots x_n$ zaporedje vhodnih podatkov s poljubno (ne vnaprej znano) dolžino n. Izhod rekurentne mreže je napovedana vrednost y_i ciljne spremenljivke glede na do sedaj videne podatke: v i-tem koraku tako rekurentna mreža napove y_i iz podatkov $x_1, x_2, \ldots, x_i$, torej realizira funkcijo $f : y_i = f(x_i, \ldots, x_2, x_1)$. Funkcija f je zaradi spremenljive dolžine



Slika 3: Shema rekurentne nevronske mreže.

vhoda v večini primerov nedosegljiva, zato njen vhod omejimo z omejitvijo na zadnjih nekaj podatkov ali pa z vpeljavo skritega vektorja h_i, ki povzema zgodovinske podatke od x_1 do x_i. V i-tem koraku tako y_i napovemo s funkcijo $y_i = f(x_i, h_{i-1})$, h_i pa izračunamo s funkcijo $h_i = g(x_i, h_{i-1})$. Ti funkciji se nevronska mreža še vedno lahko nauči s pomočjo gradientnega spusta. Shema delovanja rekurentne nevronske mreže je prikazana na sliki 3, kjer je na levi strani skrčen, rekurenten prikaz, na desni pa razpihnjen, zaporedni prikaz.



Slika 4: Shema celice GRU. Simbol $\odot$ predstavlja množenje po komponentah, $+\!\!+$ konkatenacijo vektorjev, S sigmoidno funkcijo in trapezoid polno-povezan sloj.

Zaradi numerične nestabilnosti [52] in pogosto slabših rezultatov, se namesto preproste rekurentne nevronske mreže ponavadi uporablja rekurentna nevronska mreža z vrati (GRU) ali pa nevronska mreža z dolgim kratkoročnim spominom (angl. *long short term memory, LSTM*) [23]. Celici GRU in LSTM strukturo rekurentne nevronske mreže nadgradita z mehanizmom vrat, ki se nauči kdaj je skriti vektor potrebno posodobiti in kdaj informacijo v njem pozabiti. Celica GRU uporablja dva taka mehanizma. Vrata za posodabljanje (angl. *update gate*) odločajo o količini stare informacije (h_{t-1}), ki jo bo h_t ohranil in tako poskrbijo, da gradient ne izgine ali eksplodira (numerično nestabilnost). Drugi mehanizem, imenovan vrata za resetiranje (angl. *reset gate*), pa skrbi za pozabljanje stare informacije. Shema celice GRU je prikazana na sliki 4. Vektor r_t predstavlja vrednosti, ki jih dobimo iz vrat

za resetiranje, z_t vrednosti iz vrat za posodabljanje in n_t kandidata za novo skrito stanje. Celica LSTM deluje podobno kot GRU, vsebuje pa spominska, vhodna in izhodna vrata ter dodatno stanje imenovano spomin.

2.2.2 Variacijski samokodirnik

Samokodirniki [17] so nevronske mreže, ki visoko-dimenzionalne podatke zakodirajo ali vpnejo v nizko-dimenzionalen (imenovan tudi latentni) vektorski prostor. Sestavljeni so iz kodirnika, ki podatke zakodira v latentni vektorski prostor in dekodirnika, ki podatke iz latentnega vektorskega prostora dekodira nazaj v izviren, visoko-dimenzionalen prostor. Samokodirniki se pojavijo v različnih aplikacijah [21, 22, 46] kot na primer stiskanje podatkov in odstranjevanje šuma brez poglobljene podatkovne analize. Za generiranje novih podatkov se pogosto uporabijo variacijski samokodirniki. Ti so posebna vrsta samokodirnikov, ki originalen podatek namesto v eno točko v latentnem vektorskem prostoru, zakodirajo v verjetnostno porazdelitev točk. Najpogosteje se uporabi Gaussova porazdelitev, predstavljena s pričakovano vrednostjo μ in varianco σ. Z vzorčenjem iz standardizirane Gaussove porazdelitve $\mathcal{N}(0, I)$ in dekodiranjem dobljenih vzorcev lahko variacijski samokodirnik generira nove podatke.

Variacijski samokodirniki so tesno povezani z variacijskim sklepanjem (angl. *variational inference*) [5, 3], ki se uporablja za aproksimacijo pogojne porazdelitve. V primerjavi z metodo Monte Carlo markovskih verig (angl. *Markov Chains Monte Carlo*, MCMC), ki je za tako aproksimacijo največkrat uporabljena, je variacijsko sklepanje hitrejše in bolj ustrezno za obdelavo velikih množic podatkov. V nadaljevanju uporabimo variacijsko sklepanje za izpeljavo funkcije napake z imenom variacijska spodnja meja (angl. *evidence lower bound*, ELBO), ki je za uspešno učenje variacijskega samokodirnika ključnega pomena.

Naj bo $p(Z, X) = p(Z) \cdot p(X|Z)$ skupna porazdelitev spremenljivk Z latentnega in spremenljivk X originalnega prostora. V Bayesovem modeliranju je $p(Z)$ porazdelitev latentnih spremenljivk Z in $p(X|Z)$ pogojna porazdelitev spremenljivk X ob pogoju znanih vrednosti latentnih spremenljivk Z. Sklepanje je v Bayesovem modelu pogojeno s podatki in izračunom pogojne porazdelitve latentne spremenljivke glede na podatke v originalnem prostoru, torej $p(Z|X)$. Pogojno porazdelitev $p(Z|X)$ lahko izračunamo z enačbo

$$p(Z|X) = \frac{p(Z, X)}{p(X)} = \frac{p(Z, X)}{\int p(Z, X)\, dz}. \tag{2.2}$$

Točen izračun pogojne porazdelitve $p(Z|X)$ v enačbi (2.2) je v večini primerov neobvladljiv zaradi integrala. Variacijsko sklepanje zato $p(Z|X)$ aproksimira s porazdelitvijo $q(Z) \in \tau$, kjer je τ podana družina porazdelitev. Podobnost med porazdelitvama izmerimo s Kullback-Leibler divergenco [28], ki je za porazdelitvi dveh zveznih numeričnih spremenljivk P in Q podana z enačbo

$$\mathrm{KL}(P \,\|\, Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx. \tag{2.3}$$

Želimo torej poiskati porazdelitev $q^*(Z)$ za katero velja

$$q^*(Z) = \arg\min_{q(Z)\in\tau} \mathrm{KL}(q(Z) \,\|\, p(Z|X)). \tag{2.4}$$

Neposredna optimizacija enačbe (2.4) ni mogoča, saj ne poznamo porazdelitve $p(X)$ in posledično tudi $p(Z|X)$. Ker KL divergence v enačbi (2.4) ne moremo izračunati, namesto tega optimiziramo variacijsko spodnjo mejo (ELBO), ki jo dobimo iz enačbe (2.3) in se od KL divergence razlikuje le za konstanto, kar pokažemo takole:

$$\begin{aligned}
\mathrm{KL}(q(Z) \,\|\, p(Z|X)) &= \mathbb{E}_{q(Z)}\Big[\log \frac{q(Z)}{p(Z|X)}\Big] = \\
&= \mathbb{E}_{q(Z)}\Big[\log q(Z)\Big] - \mathbb{E}_{q(Z)}\Big[\log p(Z|X)\Big] = \\
&= \mathbb{E}_{q(Z)}\Big[\log q(Z)\Big] - \mathbb{E}_{q(Z)}\Big[\log \frac{p(Z,X)}{p(X)}\Big] = \\
&= \mathbb{E}_{q(Z)}\Big[\log q(Z)\Big] - \mathbb{E}_{q(Z)}\Big[\log p(Z,X)\Big] + \mathbb{E}_{q(Z)}\Big[\log p(X)\Big].
\end{aligned} \tag{2.5}$$

Če člene enačbe (2.5) prerazporedimo, za konstanten $q(Z)$ dobimo:

$$\mathbb{E}_{q(Z)}\Big[\log p(Z,X)\Big] - \mathbb{E}_{q(Z)}\Big[\log q(Z)\Big] + \mathrm{KL}(q(Z) \,\|\, p(Z|X)) = \log p(X). \tag{2.6}$$

Z združitvijo prvih dveh členov izraza na levi strani enačbe v člen $\mathrm{ELBO}(q)$ dobimo enačbo

$$\mathrm{ELBO}(q) + \mathrm{KL}(q(Z) \,\|\, p(Z|X)) = \log p(X). \tag{2.7}$$

Ker je KL divergenca nenegativna, $\mathrm{KL}(P \,\|\, Q) \geq 0$, očitno velja $\mathrm{ELBO}(q) \leq \log p(X)$. Torej bomo z maksimizacijo variacijske spodnje meje ELBO dosegli želeni minimum KL divergence. Preden se lotimo maksimizacije ELBO, potrebujemo še naslednjo izpeljavo njegovega alternativnega izračuna:

$$\begin{aligned}
\mathrm{ELBO}(q) &= \mathbb{E}_{q(Z)}\Big[\log p(Z,X)\Big] - \mathbb{E}_{q(Z)}\Big[\log q(Z)\Big] \\
&= \mathbb{E}_{q(Z)}\Big[\log p(X|Z)\cdot p(Z)\Big] - \mathbb{E}_{q(Z)}\Big[\log q(Z)\Big] \\
&= \mathbb{E}_{q(Z)}\Big[\log p(X|Z)\Big] + \mathbb{E}_{q(Z)}\Big[\log p(Z)\Big] - \mathbb{E}_{q(Z)}\Big[\log q(Z)\Big] \\
&= \mathbb{E}_{q(Z)}\Big[\log p(X|Z)\Big] - \mathbb{E}_{q(Z)}\Big[\log \frac{q(Z)}{p(Z)}\Big] \\
&= \mathbb{E}_{q(Z)}\Big[\log p(X|Z)\Big] - \mathrm{KL}(q(Z) \,\|\, p(Z)).
\end{aligned} \tag{2.8}$$

Zadnja varianta izračuna ELBO v zgornji izpeljavi, nam pove, da maksimalno vrednost ELBO dosežemo s povečanjem uspešnosti rekonstrukcije podatkov iz njihove predstavitve v latentnem prostoru ($\log p(X|Z)$) ob hkratnem prizadevanju, da sta si porazdelitvi $q(Z)$ in $p(Z)$ čim bolj podobni.

Na osnovi zgornjih rezultatov teorije variacijskega sklepanja, lahko sestavimo arhitekturo variacijskega samokodirnika. X predstavlja podatke v originalnem prostoru, ki jih želimo z variacijskim samokodirnikom rekonstruirati, Z predstavitev

podatkov v latentnem prostoru, ki jo dobimo z uporabo kodirnika $q(Z|X)$. Nazadnje, $p(X|Z)$ predstavlja dekodirnik, ki podatke predstavljene v latentnem prostoru Z preslika nazaj v originalni prostor X. Kodirnik in dekodirnik v modelu učimo z gradientnim spustom, kjer kot ciljno funkcijo izgube uporabimo negativno vrednost ELBO, $-\text{ELBO}(q)$. Negativna vrednost poskrbi, da gradientni spust najde maksimalno vrednost ELBO.

Kot smo omenili, pogosto predpostavimo, da spremenljivke latentnega prostora Z sledijo standardizirani Gaussovi porazdelitvi $\mathcal{N}(0, I)$. Ob tej predpostavki lahko člen $\text{KL}(q(Z) \,\|\, p(Z))$ v funkciji izgube poenostavimo na

$$\text{KL}(\mathcal{N}(\mu_z, \sigma_z) \,\|\, \mathcal{N}(0, I)) = -\frac{1}{2}(1 + \log \sigma_z - \mu_z^2 - \sigma_z), \tag{2.9}$$

kjer sta μ_z in σ_z pravzaprav izhoda kodirnika $q(Z|X)$.

2.2.3 Generatorji in vpetja strukturiranih podatkov

Osnovna različica variacijskega samokodirnika torej predpostavi, da so originalni podatki vektorji v visoko-dimenzionalnem prostoru. Po drugi strani se v našem delu osredotočamo na generiranje podatkov, ki so diskretni in strukturirani. Prva pomembnejša arhitektura nevronske mreže, ki gre v smer obravnave diskretnih, strukturiranih podatkov (zaporedij simbolov) je seq2seq [41]. Slednja uporablja rekurentno nevronsko mrežo za modeliranje verjetnostne porazdelitve naslednjega simbola v zaporedju. CVAE [19] je naslednik seq2seq, ki predlaga variacijski samokodirnik za generiranje zaporedij simbolov, tudi algebrajskih izrazov. Tak pristop je sicer enostaven in učinkovit, a ne zagotavlja, da bodo generirani izrazi sintaktično pravilni.

To zagotavlja GVAE [29], ki algebrajski izraz predstavi kot zaporedje prepisovalnih pravil iz podane kontekstno neodvisne gramatike. Z uporabo variacijskega samokodirnika nato izpeljave gramatike, ki so pravzaprav zaporedja prepisovalnih pravil, zakodira v latentni prostor. GVAE tako generira izpeljave besed v podani gramatiki, ki zagotavlja, da bodo izpeljani izrazi sintaktično pravilni. Nadalje je bil GVAE v kombinaciji z Bayesovo optimizacijo v latentnem vektorskem prostoru uporabljen za odkrivanje enačb. SD-VAE [11] namesto kontekstno neodvisne gramatike uporabi atributno gramatiko in tako omogoča, da sintaktičnim omejitvam pravilnosti izrazov dodamo še semantična. Generirani izrazi so zato sintaktično in semantično pravilni.

Poleg zaporedij nizov pa lahko z globokimi generativnimi modeli proizvedemo tudi hierarhične objekte, na primer drevesa. Pristop DRNN [2] za generiranje dreves uporabi rekurentno celico, ki deluje v globino in celico, ki deluje v širino drevesa. Izhoda iz celic nato združi in napove obstoj naslednjega vozlišča (novega potomca). Namesto združitve dveh rekurentnih celic, Tree-LSTM [42] in JT-VAE [24] prilagodita rekurentno celico za hierarhične strukture. Tree-LSTM s prilagoditvijo omogoči kodiranje dreves, ki se izkaže za učinkovito pri njihovi klasifikaciji. JT-VAE je bil razvit za delo z molekulami, ki jih zakodira v dveh korakih. V prvem koraku pristop zakodira podgrafe in tako strukturo molekule spremeni v drevo, v drugem pa to drevo zakodira v latentni prostor. Z dekodiranjem v nasprotni smeri poskrbi, da so generirane molekule strukturno pravilne.

V magistrskem delu pokažemo, da lahko s podobno prilagoditvijo osnovne rekurentne celice zgradimo variacijski samokodirnik, ki je prilagojen generiranju algebrajskih izrazov. Pokazali bomo, da samokodirnik zgrajen na tej prilagoditvi učinkovito zakodira izraze v latentni prostor.

2.3 Bayesova optimizacija

Veliko algoritmov in modelov za strojno učenje vsebuje parametre, ki morajo biti za optimalno delovanje pravilno nastavljeni. Ko je takih parametrov malo, lahko njihove optimalne vrednosti iščemo z mrežo (angl. *grid search*) ali pa naključnim iskanjem (angl. *random search*). Pri velikem številu parametrov pa je takšno iskanje nepraktično, saj je za odkritje dobrih parametrov potrebno veliko število poskusov. Pogosta rešitev so metode, ki pri iskanju upoštevajo predhodno znanje o že ovrednotenih kombinacijah parametrov, na primer Bayesova optimizacija.

Bayesova optimizacija naslovi problem iskanja globalnega maksimuma x^* neznane funkcije f, torej

$$x^* = \arg\max_{x \in X} f(x), \tag{2.10}$$

kjer je X množica možnih vrednosti x parametrov funkcije f. V primerjavi s klasično optimizacijo, kjer pogosto velja $X \subseteq \mathbb{R}^n$, lahko Bayesova optimizacija preiskuje tudi prostor, ki vsebuje kategorične in pogojne vhode [37]. Za funkcijo f predpostavimo, da je „črna škatla“ brez posebne, vnaprej znane oblike. Njeno obnašanje lahko opazujemo le skozi nabor šumnih meritev y_i, ki jih pridobimo z vrednotenjem f v točkah $x_i \in X$. Vrednosti funkcije pridobivamo iterativno, zato v nadaljevanju z $D_i = \{(x_0, y_0), \ldots, (x_i, y_i)\}$ označimo pare do sedaj pridobljenih vrednosti y_i v točkah x_i.

Bayesova optimizacija iterativno gradi verjetnostni model nad prostorom X, predstavljen s točkami x in pripadajočimi meritvami vrednosti $f(x)$. V i-ti iteraciji s funkcijo izbire (angl. *acquisition function*) izbere točko $x_i \in X$, jo ovrednoti s funkcijo f in posodobi verjetnostni model. Funkcija izbire α je pomemben del pristopa, saj z uporabo verjetnostnega modela oceni potencial podane točke x_i. Posredno funkcija izbire skrbi za ravnotežje med stremenjem k lokalni izboljšavi trenutne optimalne vrednosti x in preiskovanjem širšega prostora X. Naslednjo točko za vrednotenje pridobimo z optimizacijo funkcije izbire, torej

$$x_i = \arg\max_{x \in X} \alpha(x, D_{i-1}). \tag{2.11}$$

Funkcija izbire mora biti zato računsko manj zahtevna od funkcije f: tako namreč omogoča učinkovito izbiro naslednje točke x_i z visokim potencialom, da postane nova optimalna vrednost.

Pogosto uporabljene funkcije so verjetnost izboljšave (angl. *probability of improvement*, PI), pričakovana izboljšava (angl. *expected improvement*, EI) in zgornja meja zaupanja (angl. *upper confidence bound*, UCB). Verjetnost izboljšave predpostavi, da meritve sledijo Gaussovi porazdelitvi. Ob tej predpostavki izračuna verjetnost s katero bo točka x izboljšala trenutno optimalno vrednost funkcije. Verjetnost izboljšave pogosto privede zgolj do lokalnega maksimuma, saj izračuna le verjetnost izboljšave, ne pa tudi za koliko se bo novi maksimum izboljšal. To slabost

popravi pričakovana izboljšava, ki se osredotoči na optimiziranje količine izboljšave. Nasprotno od verjetnosti izboljšave in pričakovane izboljšave, ki se osredotočita na izboljšavo, se zgornja meja zaupanja osredotoči na preiskovanje prostora z visoko negotovostjo. Zaradi tega je zgornja meja zaupanja pogosto bolj primerna za preiskovanje visoko dimenzionalnega prostora kot preostali dve funkciji.

V magistrski nalogi z Bayesovo optimizacijo pospešimo preiskovanje latentnega vektorskega prostora, v katerega so izrazi zakodirani. Tako se izognemo slabosti verjetnostnih pristopov kot je ProGED [6], v katerih se izrazi vzorčijo naključno.

3 Metode

V tem poglavju predstavimo izrazna drevesa in strukturo našega modela. Poleg tega opišemo tudi postopek učenja, generiranja novih podatkov in preiskovanja latentnega vektorskega prostora z Bayesovo optimizacijo.

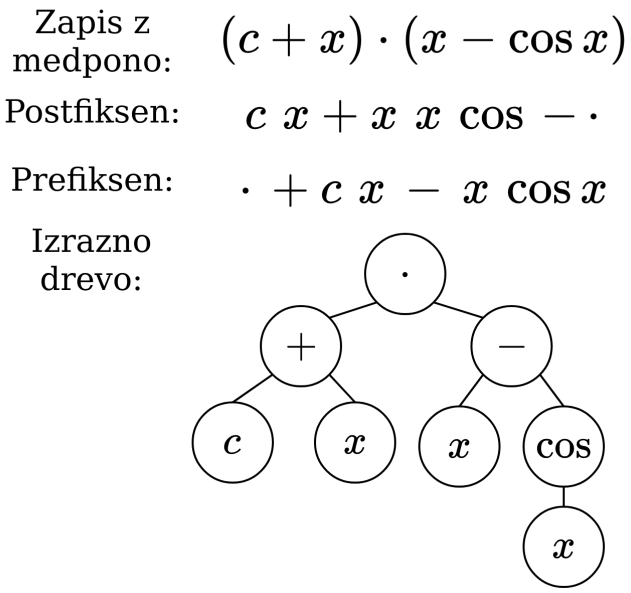
3.1 Predstavitev izrazov

Najpogosteje izraz predstavimo kot niz znakov v zapisu z medpono (angl. *infix notation*). V tem zapisu se operator nahaja med dvema manjšima izrazoma, podizrazoma A in B (na primer $A+B$). Prednost tega zapisa je berljivost, saj je že na prvi pogled očitno nad katerima podizrazoma je določen operator uporabljen. Slabost zapisa pa je potreba po oklepajih, ki določijo vrstni red operacij, saj ti izraz podaljšajo in ga naredijo bolj kompleksnega. Temu problemu se lahko izognemo z uporabo Poljskega (angl. *prefix*) ali obrnjenega (angl. *postfix*) zapisa. V Poljskem zapisu se operator nahaja pred podizrazoma (na primer $+AB$), v obrnjenem pa za njima (na primer $AB+$). Obrnjen zapis se pojavi predvsem kot vmesni zapis programske kode izrazov v prevajalnikih, saj omogoča hiter in preprost izračun vrednosti izraza z uporabo sklada.

Na zapise niza lahko gledamo kot različne načine sprehajanja po izraznem drevesu, kjer Poljski zapis ustreza prememu, zapis z medpono vmesnemu, obrnjen zapis pa obratnemu sprehodu. Izrazno drevo je dvojiško drevo v katerem so konstante in spremenljivke v listih, operatorji in funkcije pa v notranjih vozliščih. Vozlišča z operatorji imajo dva naslednika, tista s funkcijami pa enega. Primer izraznega drevesa, skupaj z različnimi zapisi izraza $(c+x)\cdot(x-\cos x)$, je predstavljen na sliki 5. Vidimo lahko, da so v zapisu z medpono potrebni oklepaji, saj bi se po dogovoru operator množenja izvedel pred operatorjema za seštevanje in odštevanje.

Čeprav številni pristopi strojnega učenja in generativni modeli, zaradi pogoste uporabe na področju obdelave naravnega jezika, omogočajo delo z nizi, predstavitev izrazov z nizi ni najbolj primerna. Glavni razlog za to je, da je struktura (sintaksa) izrazov, za razliko od strukture naravnega jezika, zelo strogo določena. Že en napačen simbol v nizu lahko naredi izraz nepravilen in neuporaben za nadaljnjo uporabo oziroma izračun. Primera takih nizov v zapisu z medpono sta „$(1+x)\cdot+$“ in „$)1+x)\cdot 9$“, v Poljskem zapisu „$9\cdot+1x$“ in „$\cdot++1x$“, v obrnjenem zapisu pa „$1x++\cdot$“.

Z generiranjem izraznih dreves namesto nizov se lahko opisanemu problemu izognemo, saj za vsako vozlišče izraznega drevesa vemo koliko naslednikov potrebuje.



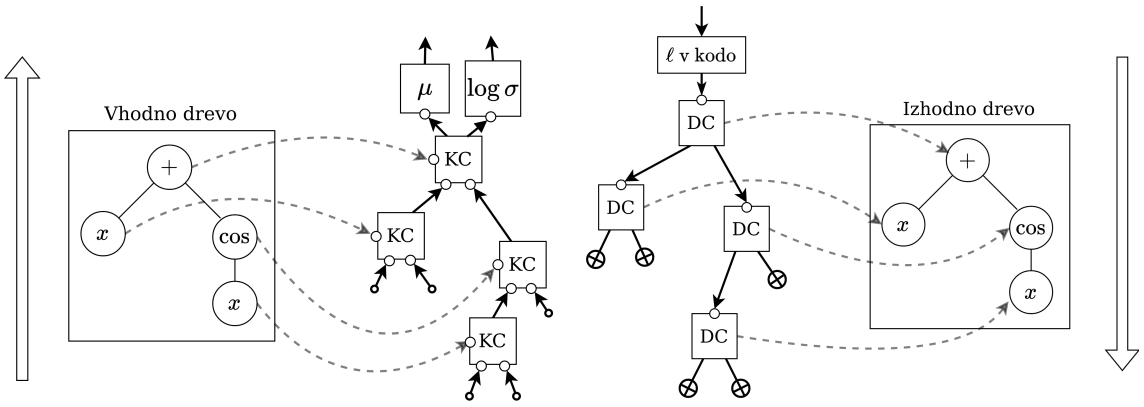
Slika 5: Primer izraza v različnih predstavitvah

Generirana izrazna drevesa tako vedno podajo sintaktično pravilen izraz. Generiranje izraznih dreves pa ima še druge prednosti. Razumno je pričakovati, da se enaki podizrazi (na primer $x + 1$) vedno zakodirajo enako ne glede na njihovo pozicijo v izrazu, saj so podizrazi neodvisni od preostalega izraza in njihove pozicije. Pri kodiranju izraznega drevesa lahko zagotovimo kodiranje, ki upošteva to pričakovanje, pri kodiranju niza pa tega ne moremo zagotoviti, saj je kodiranje odvisno tudi od konteksta podizraza v nizu, torej simbolov, ki so se pojavili pred podizrazom. *Nenazadnje*, pri dekodiranju niza, začetna informacija potuje n korakov, kjer je n dolžina niza, medtem ko je pri dekodiranju izraznega drevesa dolžina poti informacije skrajša na $\log n$. Zaradi tega se pri dekodiranju niza izgubi veliko več informacije in je tako rekonstrukcija začetnega niza težja.

3.2 Predstavitev generativnega modela

Generativni model za izraze je zasnovan na arhitekturi variacijskega samokodirnika. Kodirnik zakodira izrazno drevo v vektor konstantne dolžine, ki ga imenujemo latentna koda, dekodirnik pa latentno kodo dekodira v izrazno drevo. H generiranju izraznih dreves lahko nato pristopimo tako, da generiramo naključne vektorje konstantne dolžine, ki jih dekodirnik nato prevede v izrazno drevo.

Osnovna enota kodirnika je kodirna celica (KC), ki sledi strukturi izraznega drevesa in kodira njegova vozlišča. Kodiranje se začne v listih, za tem pa se rekurzivno zakodirajo še notranja vozlišča, vse do korena. Kodirna celica na vhod prejme simbol vozlišča (operator, funkcijo, spremenljivko ali konstanto) in kodi njegovih naslednikov, na izhod pa vrne njegovo kodo. Kjer vozlišče nima dveh naslednikov (listi in funkcije) se namesto kode naslednika v kodirno celico pošlje ničeln vektor. Po kodiranju se koda korena pošlje čez polno-povezana sloja μ in $\log \sigma$, ki generirata izhod iz kodirnika. Primer kodiranja izraz $x + \cos x$ je prikazan na sliki 6a. Najprej se



(a) Kodiranje (b) Dekodiranje

Slika 6: Proces kodiranja in dekodiranja izraza $x+\cos x$. Kratici KC in DC ustrezata terminom „kodirna celica“ (predstavljena v poglavju 3.2.1) in „dekodirna celica“ (predstavljena v poglavju 3.2.2).

zakodirata vozlišči s spremenljivko x. Kodirna celica tako na vhod dobi simbol x in dva ničelna vektorja, saj vozlišči nimata naslednika. Nato gre čez kodirno celico vozlišče s funkcijo cos. Ta ima enega naslednika, zato kodirna celica na vhod prejme simbol cos, kodo naslednika x in ničeln vektor. Nazadnje se koren drevesa zakodira v vektor k. Tu kodirna celica na vhod prejme simbol + in kodi naslednikov x in $\cos x$. Kodiranje se zaključi tako, da vektor k pošljemo čez polno-povezana sloja μ in $\log\sigma$, ki na izhod vrneta vektorja μ_ℓ in $\log\sigma_\ell$.

Pri generiranju izrazov bomo latentne kode generirali z vzorčenjem iz Gaussove porazdelitve. Slednjo parametriziramo z vektorjema μ_ℓ in $\log\sigma_\ell$, ki sta izhoda kodirnika. Ker nevronska mreža ne more naključno vzorčiti iz porazdelitve, uporabimo trik reparametrizacije. Neodvisno od nevronske mreže vzorčimo ϵ iz standardizirane Gaussove porazdelitve in ga nato predelamo v latentno kodo izraznega drevesa $\tilde{\ell}$ z enačbo $\tilde{\ell} = \mu_\ell + \sigma_\ell \odot \epsilon$, kjer $\odot$ predstavlja množenje vektorjev po komponentah.

Dekodirnik najprej latentno kodo pretvori v kodo korena izraznega drevesa s polno povezanim slojem. V drugem delu nato s pomočjo dekodirne celice (DC) rekurzivno generiramo izrazno drevo. Dekodirna celica na vhod prejme kodo vozlišča, na izhod pa vrne kodi naslednikov in simbol trenutnega vozlišča. Nasledniki vozlišča se nato na isti način dekodirajo glede na dobljen simbol. Če je dobljen simbol operator, se rekurzivno dekodirata oba naslednika, če je funkcija, se dekodira le en (levi) naslednik, v primeru spremenljivke ali konstante, pa se rekurzija in s tem dekodiranje ustavi.

Primer dekodiranja izraza $x + \cos x$ je prikazan na sliki 6b. Najprej pošljemo latentna koda $\tilde{\ell}$ čez polno-povezan sloj in jo s tem pretvori v kodo korena. To kodo nato pošljemo v dekodirno celico, ki proizvede simbol + in kodi naslednikov. Generiranje naslednikov se nadaljuje, saj ima operator + dva naslednika. Z dekodirno celico in kodo levega naslednika nato proizvedemo simbol x. Spremenljivke nimajo naslednikov, zato se generiranje na tej veji se ustavi. Ko čez dekodirno celico pošljemo kodo desnega naslednika korena dobimo simbol cos in dva naslednika. Kosinus je funkcija, zato se generiranje nadaljuje le v levem sosedu. Koda levega soseda

kosinusa proizvede simbol x. S tem se generiranje drevesa konča, saj se rekurzija v vseh vejah zaključi.

3.2.1 Kodirnik in kodirna celica

Kodirnik je v variacijskih samokodirnikih sestavljen iz dveh delov. Prvi del podatke zakodira v srednjo ali skrito kodo (angl. *hidden code*). Strukturo tega dela kodirnika prilagajamo tipu podatkov. Tako, na primer, za tabelarične podatke uporabljamo polno povezane sloje, za slike kombinacijo konvolucijskih in akumulacijskih slojev, za nize pa osnovne celice rekurentnih mrež. Drugi del skrito kodo pretvori v vektorje, ki predstavljajo parametre izbrane (največkrat Gaussove) porazdelitve. Struktura tega dela kodirnika se ponavadi ne razlikuje med različnimi samokodirniki.

Bistvena razlika med drevesi in nizi je število naslednikov vozlišča oziroma simbola. Simbol v nizu ima namreč enega naslednika, medtem ko ima vozlišče v dvojiškem drevesu dva. Osnovno celico rekurentne nevronske mreže z vrati (angl. *gated recurrent unit, GRU*) [10], ki se velikokrat uporablja v prvem delu kodirnika nizov, smo zato prilagodili v novo različico GRU221. Slednja na vhodu sprejme dve skriti kodi obeh naslednikov vozlišča, namesto le ene, ki jo sprejme običajna različica GRU. Koda nadrejenega vozlišča h je tako izračunana iz vhodnega simbola x ter skritih kod vozlišč naslednikov h_l in h_r z enačbami:

$$r = S(W_{ir}x + b_{ir} + W_{hr}(h_l +\!\!+ h_r) + b_{hr}) \tag{3.1}$$

$$z = S(W_{iz}x + b_{iz} + W_{hz}(h_l +\!\!+ h_r) + b_{hz}) \tag{3.2}$$

$$n = \tanh(W_{in}x + b_{in} + r \odot (W_{hn}(h_l +\!\!+ h_r) + b_{hn})) \tag{3.3}$$

$$h = (1 - z) \odot n + \frac{z}{2} \odot h_l + \frac{z}{2} \odot h_r, \tag{3.4}$$

kjer je S sigmoidna funkcija $S(u) = 1/(1 + \exp(-u))$, $h_l +\!\!+ h_r$ pa stik (angl. *concatenation*) vektorjev h_l in h_r. Med enačbami za GRU221 in GRU sta dve pomembni razliki. GRU uporabi le skrito kodo prejšnjega simbola, medtem ko GRU221 uporabi stik dveh skritih kod podrejenih vozlišč h_l in h_r. Posledično se vhodna dimenzija matrik z utežmi (W_{hr}, W_{hz} in W_{hn}) spremeni iz $\dim(h)$ v $\dim(h_l)+\dim(h_r) = 2\cdot\dim(h)$. Druga razlika se pojavi v enačbi (3.4), kjer je drugi del, ki je v GRU enoti enak $z \odot h_{n-1}$, spremenjen v $\frac{z}{2} \odot h_l + \frac{z}{2} \odot h_r$, da ohrani informacijo iz skritih kod obeh naslednikov.

Tako spremenjena struktura celice GRU221 omogoča kodiranje drevesa ne da bi ga morali pretvoriti v niz s sprehodom, kar bi bilo ekvivalentno temu, da uporabljamo predstavitev izrazov z nizi. Prednost takega neposrednega kodiranja drevesa je, da se enaki podizrazi zakodirajo v enake kodirne vektorje, ne glede na njihov položaj v izrazu. Posledično lahko pričakujemo, da bo kodirnik dobro rekonstruiral izraze, ki jih ni videl med učenjem samokodirnika. Celo več, pričakujemo lahko, da bo samokodirnik omogočal dobro rekonstrukcijo izrazov tudi iz manjših učnih množic in ob uporabi latentnega prostora nižjih razsežnosti.

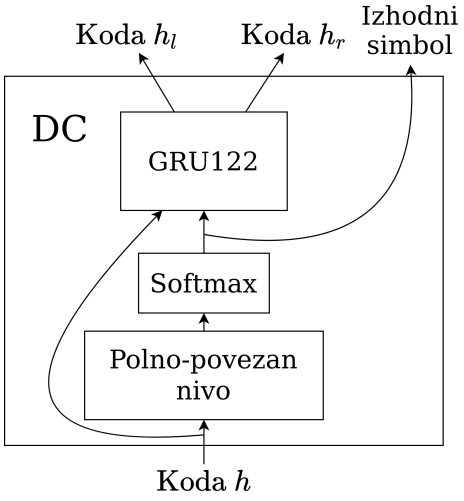
Posebnost variacijskih samokodirnikov je latentna koda, ki predstavlja porazdelitev. Kot v najpogostejšem primeru, tudi mi za latentno kodo uporabimo Gaussovo porazdelitev parametrizirano z vektorjem pričakovanih vrednosti μ_ℓ in vektorjem varianc σ_ℓ. Vektor, ki predstavlja pričakovane vrednosti dobimo tako, da kodo korenskega vozlišča izraznega drevesa pošljemo čez polno povezan sloj. Pri varianci

je potrebno še dodatno poskrbeti, da so vse komponente kodirnega vektorja nenegativne. To zagotovimo tako, da pošljemo vmesno kodo čez polno povezan sloj in njegove izhode interpretiramo kot logaritme komponent vektorja varianc σ_ℓ.

3.2.2 Dekodirnik in dekodirna celica

Drugi del samokodirnika je dekodirnik, ki latentno kodo pretvori v izrazno drevo. Dekodirnik je sestavljen iz polno-povezanega sloja, ki latentno kodo spremeni v kodo korena izraznega drevesa $\tilde{\ell}$ in dekodirne celice, ki kodo vozlišča pretvori v izhodni simbol in kodi naslednikov.

Struktura dekodirne celice je prikazana na sliki 7. Celica na vhod prejme kodo vozlišča (označeno s h), ki gre najprej čez polno-povezan, nato pa čez sloj tipa softmax. Vektor, ki ga s tem dobimo predstavlja porazdelitev verjetnosti za izhodne simbole in ima v celici dve vlogi. Simbol z največjo vrednostjo izberemo kot izhodni simbol oziroma oznako trenutnega vozlišča izraznega drevesa, poleg tega pa vektor porazdelitev uporabimo kot vhod v GRU122 celico. GRU122 na vhod, poleg omenjenega vektorja porazdelitev, prejme tudi kodo vozlišča h, na izhod pa vrne kodi za levega in desnega naslednika (h_l in h_r). Dekodirna celica ima tako tri izhode: oznako trenutnega vozlišča v izraznem drevesu ter kodi h_l in h_r za levega in desnega naslednika.



Slika 7: Struktura dekodirne celice

Podobno kot GRU221, je tudi GRU122 prilagoditev običajne različice celice GRU, ki na vhod prejme kodo vozlišča h in vektor porazdelitev verjetnosti za izhodne simbole x, na izhod pa vrne kodi levega in desnega naslednika. Ti kodi dobimo

z enačbami:

$$r = S(W_{ir}x + b_{ir} + W_{hr}h + b_{hr}) \tag{3.5}$$
$$z = S(W_{iz}x + b_{iz} + W_{hz}h + b_{hz}) \tag{3.6}$$
$$n = \tanh(W_{in}x + b_{in} + r \odot (W_{hn}h + b_{hn})) \tag{3.7}$$
$$l = (1 - z) \odot n + z \odot (h +\!\!+ h) \tag{3.8}$$
$$l \equiv h_l +\!\!+ h_r. \tag{3.9}$$

Tudi med enačbami za GRU122 in enačbami za GRU sta dve razliki. Vektorji r, z, n in l so dimenzije $2 \cdot \dim(h)$. Posledično je dimenzija vektorjev odmikov in izhodna dimenzija matrik uteži $2 \cdot \dim(h)$ namesto običajne $\dim(h)$. Druga razlika je v enačbi (3.8), kjer s stikom dveh kopij kode vozlišča h poskrbimo, da se dimenzije vektorjev ujemajo. Ekvivalenca (3.9) nam pove, da vektor l interpretiramo kot stik kod naslednikov h_l in h_r, torej ga lahko razdelimo na dva enako dolga vektorja in dobimo želeni kodi.

3.3 Učenje

Globoke nevronske mreže vsebujejo uteži in odmike (parametre), katerih vrednosti se je pred uporabo potrebno naučiti iz podatkovne množice. Pri učenju parametrov samokodirnikov minimiziramo rekonstrukcijsko napako, t.j., razliko med podanim učnim primerom (izraznim drevesom) in rekonstrukcijo le-tega, generiranega s samokodirnikom. Učenje parametrov je lahko pri variacijskih samokodirnikih zelo zahtevno, saj se ob nepravilnem procesu učenja, model rekonstrukcije ne nauči. Pri učenju je zato potrebno model dodatno usmerjati in s tem nalogo učenja omejiti in poenostavi. V našem pristopu model med učenjem usmerimo tako, da pri rekonstrukciji ohranimo strukturo podanega, učnega izraznega drevesa in se osredotočimo le na pravilne napovedi simbolov v vozliščih. To ne vpliva na rekonstrukcijo strukture drevesa, saj je le-ta določena s simboli, ki se v vozliščih pojavijo. Med učenjem samokodirnika torej nadaljujemo z generiranjem obeh naslednikov vozlišča, če tako narekuje struktura učnega primera, četudi model kot oznako vozlišča napove konstanto c in ne aritmetičnega operatorja.

Zaradi ohranjanja začetne strukture drevesa imata tako začetni kot tudi rekonstruiran izraz enako strukturo izraznega drevesa. S premim sprehodom čez izrazno drevo torej dobimo zaporedje, kjer se istoležna vozlišča nahajajo na enakih mestih. Iz začetnega izraza tako dobimo zaporedje ciljnih simbolov, iz rekonstruiranega izraza pa zaporedje vektorjev, ki ustrezajo porazdelitvam verjetnosti simbolov v vsakem vozlišču. Iz teh zaporedji izračunamo napako rekonstrukcije, ki je sestavljena iz dveh členov. Prvi ustreza napaki rekonstrukcije CE (angl. *cross-entropy loss*), ki jo izračunamo s pomočjo prečne entropije med učnim primerom in zaporedjem vektorjev, ki ustrezajo porazdelitvam. Drugi člen ustreza Kullback-Leiblerjevi divergenci [28] (KL), ki poskrbi za regularizacijo (angl. *regularization*) napake.

Funkcija izgube na osnovi prečne entropije se pogosto uporablja v klasifikaciji, saj ta velikokrat privede do hitrejšega učenja in boljšega prileganja podatkom. Podana

je s formulo:

$$\mathrm{CE}(p,q) = \frac{1}{N}\sum_{i=1}^{N}\left(-\sum_{j=1}^{C} p_{ij}\log q_{ij}\right), \tag{3.10}$$

kjer je p_i porazdelitev verjetnosti simbolov v i-tem vozlišču podanega učnega primera (verjetnost p_{ij} je 1 zgolj takrat, ko v i-tem vozlišču opazujemo j-ti simbol, in 0 sicer), q_i porazdelitev verjetnosti simbolov za i-to vozlišče rekonstruiranega izraznega drevesa, C število različnih simbolov (oznak vozlišč) ter N število simbolov v izrazu, ki ustreza drevesu.

Drugi del funkcije napake je Kullback-Leiblerjeva divergenca, ki se pogosto uporablja v variacijskih samokodirnikih, saj poskrbi za regularizacijo rekonstrukcijske napake tako, da omejuje magnitudo kodirnega vektorja. KL divergenco izračunamo iz μ_ℓ in $\log\sigma_\ell$, ki jih dobimo iz kodirnika s funkcijo:

$$\mathrm{KL}(\mu,\log\sigma) = -\frac{1}{2}\sum_{i=1}^{\dim(\mu)}\left(1+\log\sigma_i-\mu_i^2-\sigma_i\right). \tag{3.11}$$

Celotna funkcija napake je torej:

$$\mathrm{E}(x,y,\mu_\ell,\log\sigma_\ell) = \mathrm{CE}(x,y) + \alpha\cdot\mathrm{KL}(\mu_\ell,\log\sigma_\ell), \tag{3.12}$$

kjer je $\alpha\in[0,\infty]$ parameter, ki določa stopnjo regularizacije. Visoke vrednosti tega parametra usmerijo učenje v minimizacijo magnitude latentne kode izraznih dreves, nizke vrednosti pa v minimizacijo napake rekonstrukcije.

Med učenjem uporabljamo ohlajanje (angl. *annealing*) parametra α, saj na ta način model usmerimo, da se ob začetku učenja učenja osredotoči na dobro rekonstrukcijo, šele nato pa na regularizacijo [3]. Parameter α na začetku učenja nastavimo na vrednost $1.2\cdot10^{-4}$, nato pa vrednost parametra prvih 2500 iteracij posodabljamo z izrazom $\alpha_i = \frac{\tanh((i-4500)\cdot0.001)+1}{2}$, kjer je α_i vrednost parametra α v i-ti iteraciji.

Število parametrov modela je odvisno od dimenzije latentne kode (označene z β), dimenzije skrite kode (označene z γ) in števila različnih simbolov, ki se lahko v izrazu pojavijo (označenim s C). Kodirnik je sestavljen iz dveh polno-povezanih slojev in enote GRU221. Polno-povezana sloja vsebujeta $2\cdot\beta\cdot(\gamma+1)$ parametrov, saj sta sestavljena iz dveh matrik dimenzije $\beta\times\gamma$ in dveh odmikov velikosti $1\times\beta$. Enota GRU221 vsebuje tri matrike dimenzije $C\times\gamma$, tri matrike dimenzije $2\cdot\gamma\times\gamma$ in šest odmikov dimenzije $1\times\gamma$, skupaj $3\cdot\gamma\cdot(C+2\cdot(\gamma+1))$ parametrov. Kodirnik ima torej skupaj

$$\begin{aligned} \mathit{par}(\mathit{kodirnik}) &= 2\cdot\beta\cdot(\gamma+1)+3\cdot\gamma\cdot(C+2\cdot(\gamma+1)) = \\ &= \gamma\cdot(2\cdot\beta+6\cdot\gamma+3\cdot C+6)+2\cdot\beta \end{aligned}$$

paramatrov. Dekodirnik je sestavljen iz enega polno-povezanega sloja s $\gamma\cdot(\beta+1)$ parametri in dekodirno enoto. Ta vsebuje polno-povezan sloj s $C\cdot(\gamma+1)$ parametri in enoto GRU122. Enota GRU122 vsebuje tri matrike dimenzije $C\times2\cdot\gamma$, tri matrike dimenzije $\gamma\cdot2\times\gamma$ in šest odmikov dimenzije $1\times2\cdot\gamma$, skupaj $6\cdot\gamma\cdot(C+\gamma+2)$

parametrov. Dekodirnik ima torej

$$\begin{aligned} par(dekodirnik) &= \gamma \cdot (\beta + 1) + C \cdot (\gamma + 1) + 6 \cdot \gamma \cdot (C + \gamma + 2) = \\ &= \gamma \cdot (\beta + \gamma + 7 \cdot C + 13) + C \end{aligned}$$

parametrov, celoten model pa

$$\begin{aligned} par(model) &= par(kodirnik) + par(dekodirnik) = \\ &= \gamma \cdot (3 \cdot \beta + 7 \cdot \gamma + 10 \cdot C + 19) + 2 \cdot \beta + C. \end{aligned}$$

Model z $\beta = \gamma = 128$ in $C = 7$ ima v tem primeru 175495 parametrov, model z $\beta = \gamma = 32$ in $C = 7$ pa 12159.

3.4 Generiranje izrazov

Naš pristop je uporaben predvsem za generiranje izrazov. Za raziskovanje prostora izrazov in iskanje tistega, ki najbolj ustreza podatkom lahko uporabimo naključno generiranje izrazov. Za to zadostuje dekodirnik z naučenimi parametri. Naključen izraz generiramo tako, da iz standardizirane Gaussove porazdelitve $\mathcal{N}(0, I)$ vzorčimo vektorje, ki jih nato pošljemo čez dekodirnik.

Poleg naključnega, naš pristop omogoča tudi generiranje izrazov, ki so podobni prej izbranemu izrazu.To želimo na primer, ko vemo, da lahko ciljno spremenljivko aproksimiramo z enačbo $x^2 + 1$, a se ta ne prilega povsem. S pristopom tako generiramo enačbe, ki so podobne $x^2 + 1$ in ugotovimo, da je iskana enačba $1.1 \cdot x^2 + 1$. Ta način geneiranja izrazov je torej uporaben v primerih, ko želimo že poznano enačbo prilagoditi novim podatkom. Za generiranje takih izrazov je pomembno, da je latentni prostor, ki se ga samokodirnik nauči zvezen, torej da sta si podobna izraza v latentnem prostoru blizu. Podobne izraze generiramo tako, da najprej začetni izraz pošljemo čez kodirnik in tako dobimo μ_ℓ in $\log \sigma_\ell$. Nato iz Gaussove porazdelitve s centrom v μ_ℓ in varianco $e^{\log \sigma_\ell} = \sigma_\ell$, torej $\mathcal{N}(\mu_\ell, \sigma_\ell)$, vzorčimo vektorje in jih pošljemo čez dekodirnik. Pričakujemo, da bodo zaradi zveznosti latentnega prostora tako generirani izrazi (sintaktično) podobni začetnemu.

3.5 Uporaba Bayesove optimizacije

V magistrskem delu uporabimo Bayesovo optimizacijo za odkrivanje enačb. Z Bayesovo optimizacijo namreč preiskujemo latentni prostor izrazov, ki ga proizvede prej opisani variacijski samokodirnik. Ob podanem variacijskem samokodirniku je naš pristop za odkrivanje enačb sledeč. Najprej iz standardizirane Gaussove porazdelitve $\mathcal{N}(0, I)$ vzorčimo začetne vektorje oziroma vrednosti x_i. Te vektorje pošljemo čez dekodirnik in tako pridobimo strukture izrazov na desni strani enačb.

Nato z algoritmom diferencialne evolucije (angl. *differential evolution*) [40] poiščemo optimalne vrednosti konstant v strukturah izrazov in z njimi izračunamo MSE napako za posamezen izraz. Ta korak dodatno oteži odkrivanje enačbe, saj je iskanje optimalnih vrednosti konstant problem numerične optimizacije, ki pogosto ne najde najboljših oziroma točnih vrednosti.

V nadaljnjih iteracijah Bayesove optimizacije z vektorji x_i in pripadajočimi napakami y_i naučimo model Gaussovega procesa [33], ki ga uporablja Bayesova optimizacija. Z zanko nato iterativno vzorčimo nove vektorje, iz njih generiramo strukture izrazov, strukturam določimo konstante, izračunamo napako teh izrazov in Gaussov proces nadgradimo z novimi učnimi primeri. Ob koncu zanke izberemo izraz z najnižjo napako MSE. Naloga odkrivanja enačb je uspešna, če je napaka najboljšega izraza pod vnaprej določeno mejo (na primer 10^{-20}).

Izrazi so zaradi regularizacijie s KL divergenco v latentnem prostoru porazdeljeni s standarizirano Gaussovo porazdelitvijo $\mathcal{N}(0, I)$. Iz te porazdelitve za boljšo konvergenco Bayesove optimizacije pri vzorčenju uporabimo zaporedja Sobol [39]. Za funkcijo izbire pa uporabimo pričakovano izboljšavo.

Med vrednotenjem smo za implementacijo opisanega postopka uporabili Python knjižnico BOTorch [4]. Za optimalno delovanje modelov v knjižnici mora biti vrednost, ki jo želimo maksimizirati v mejah $[0, 1]$. V našem pristopu zato optimiziramo vrednost, ki jo dobimo s formulo

$$L(x) = \frac{2}{1 + e^{kx}}, \tag{3.13}$$

kjer x predstavlja MSE napako izraza in k parameter izračunan iz podatkov. Če z y_{best} označimo najnižji MSE izrazov iz začetne množice, k izračunamo z izrazom $k = \frac{\log 3}{y_{\text{best}}}$.

4 Rezultati

V tem poglavju predstavimo rezultate empiričnega vrednotenja našega pristopa. Najprej predstavimo delež sintaktično nepravilnih izrazov in rekonstrukcijsko napako pristopov. Nato raziščemo vpliv velikosti učne množice in dimenzije latentnega vektorskega prostora na rekonstrukcijo. Za tem z linearno interpolacijo med izrazi prikažemo, da se podobni izrazi zakodirajo v točke, ki so si v latentnem vektorskem prostoru blizu. Nazadnje z uporabo Bayesove optimizacije prikažemo nekaj primerov odkrivanja enačb.

4.1 Okvir empiričnega vrednotenja

V nadaljevanju naš pristop empirično ovrednotimo na različnih podatkovnih množicah in ga primerjamo s podobnimi pristopi. V tem podpoglavju zato opišemo uporabljene podatkovne množice, parametre uporabljenih pristopov in metrike za primerjalno vrednotenje pristopov. Naš pristop je dostopen na repozitoriju `https://github.com/brencej/ProGED`, pristopa CVAE in GVAE pa sta prilagojena po repozitoriju `https://github.com/mkusner/grammarVAE`.

4.1.1 Podatkovne množice

Pri vrednotenju smo uporabili šest sintetično generiranih množic izrazov z različnimi lastnostmi. Podatkovne množice lahko razdelimo v dve skupini; tiste brez funkcij (sinus in kosinus) in tiste z njimi. Vsaka od teh skupin vsebuje majhno podatkovno

množico kratkih izrazov, veliko množico kratkih izrazov in veliko podatkovno množico dolgih izrazov. Maksimalno dolžino izrazov znotraj množice uravnavamo tako, da omejimo maksimalno višino dreves, ki se v množici pojavijo. Višino drevesa definiramo kot število vozlišč na najdaljši poti med listi in korenom drevesa. Bolj podrobni opisi podatkovnih množic so naslednji:

AE4-2k vsebuje 2,000 izrazov z izraznimi drevesi višine največ 4. Ti izrazi vsebujejo konstante c, spremenljivke x, in operatorje $+,-,\cdot,/,\hat{}$.

Trig4-2k je podobna podatkovni množici AE4-2k, a dodatno vsebuje izraze s trigonometričnima funkcijama sinus in kosinus.

AE5-15k vsebuje 15,000 izrazov z izraznimi drevesi višine največ 5. Ti izrazi vsebujejo konstante c, spremenljivke x, in operatorje $+,-,\cdot,/,\hat{}$.

Trig5-15k je podobna podatkovni množici AE5-15k, a dodatno vsebuje izraze s trigonometričnima funkcijama sinus in kosinus.

AE7-20k vsebuje 20,000 izrazov z izraznimi drevesi višine največ 7. Ti izrazi vsebujejo konstante c, spremenljivke x, in operatorje $+,-,\cdot,/,\hat{}$.

Trig7-20k je podobna podatkovni množici AE7-20k, a dodatno vsebuje izraze s trigonometričnima funkcijama sinus in kosinus.

Izraze v podatkovnih množicah generiramo na sledeč način. Najprej s pomočjo Python knjižnice ProGED[1] [6] generiramo izraze v zapisu z medpono. Pri tem za podatkovne množice brez funkcij uporabimo verjetnostno gramatiko:

$$
\begin{aligned}
&S \to SAF\,[0.4] \mid F\,[0.6]\\
&A \to +\,[0.5] \mid -\,[0.5]\\
&F \to FBT\,[0.4] \mid T\,[0.6]\\
&B \to \cdot\,[0.5] \mid /\,[0.5]\\
&T \to (S)\,[0.25] \mid c\,[0.375] \mid x\,[0.375],
\end{aligned}
$$

v množicah s sinusom in kosinusom pa dodamo nov ne-terminal L in ne-terminal T spremenimo na sledeč način:

$$
\begin{aligned}
&T \to (S)\,[0.15] \mid \cos(S)\,[0.05] \mid \sin(S)\,[0.05] \mid L\,[0.75]\\
&L \to c\,[0.5] \mid x\,[0.5].
\end{aligned}
$$

Generirane izraze nato poenostavimo s pomočjo Python knjižnjice SymPy [32] in tako dobimo operator za potenciranje, ki ga v gramatikah ni. Nazadnje izraze pretvorimo v izrazna drevesa s pomočjo algoritma ranžirne postaje (angl. *shunting yard algorithm*) [12].

4.1.2 Parametri preizkušenih pristopov

Med empiričnim vrednotenjem naš pristop (HVAE, opisan v poglavju 3) primerjamo z GVAE [29] in CVAE [19] (predstavljena v poglavju 2.2.3). Pristopa GVAE

[1] Dostopno na `https://github.com/brencej/ProGED`

in CVAE treniramo 150 epoh s podatki združenimi v skupine velikosti 64, dimenzijo latentnega prostora 128 in velikostmi konvolucijskih filtrov 2, 3 in 4. HVAE treniramo 20 epoh z dimenzijo latentnega prostora 128 in podatki združenimi v skupine velikosti 32.

4.1.3 Ocena rekonstrukcijske napake

Učinkovitost samokodirnikov pogosto merimo z rekonstrukcijsko napako. To je na izrazih težko meriti, saj le-ti vsebujejo komutativne operatorje in v zapisu z medpono nepotrebne oklepaje. Zgornjo mejo rekonstrukcijske napake lahko ocenimo z uporabe Levenshteinove razdalje [31], ki je pogosto imenovana tudi razdalja urejanja (angl. *edit distance*). Podana je s formulo

$$r(a,b) = \begin{cases} |b| & |a| = 0 \\ |a| & |b| = 0 \\ 1 + \min \begin{cases} r(\mathrm{rep}(a), b) \\ r(a, \mathrm{rep}(b)) \\ r(\mathrm{rep}(a), \mathrm{rep}(b)) \end{cases} & \text{sicer,} \end{cases} \tag{4.1}$$

kjer $|a|$ predstavlja dolžino niza a, klic funkcije $\mathrm{rep}(x)$ pa vrne niz x brez prvega simbola. Intuitivno razdalja urejanja pove koliko vstavljanj novih ter odstranitev in sprememb obstoječih simbolov je potrebnih, da iz enega niza dobimo drugega. Razdalja urejanja se pogosto uporablja v biologiji za iskanje podobnosti med DNK zapisi in v procesiranju naravnega jezika pri popravljanju napačno črkovanih besed in približnem iskanju nizov.

Rekonstrukcijo izrazov testiramo na sledeč način. Najprej izraz pošljemo čez samokodirnik in ga rekonstruiramo. Nato po potrebi preverimo sintaktično pravilnost rekonstruiranega izraza in ga spremenimo v izrazno drevo. Z obratnim sprehodom čez drevo za tem generiramo izraz v obrnjenem zapisu. Nazadnje izračunamo rekonstrukcijsko napako tako, da izračunamo razdaljo urejanja med originalnim in rekonstruiranim izrazom v obrnjenem zapisu.

Izraze primerjamo v obrnjenem zapisu, saj ta zapis ne potrebuje oklepajev. Oklepaji izrazu namreč dodajo nepotrebne simbole, ki izraz podaljšajo, ne pa nujno spremenijo. Napaka je ob prisotnosti oklepajev lahko neničelna, čeprav se izraza semantično ne razlikujeta. Primer tega sta izraza „$(1+x)\cdot 9$“ in „$(((1+x))\cdot 9)$“ med katerima je razdalja urejanja 4.

Zaradi komutativnosti operatorjev $+$ in $\cdot$, razdalja urejanja predstavlja zgornjo mejo napake. V primeru, da sta podizraza v pravilnem vrstnem redu je razdalja pravilno izračunana, v primeru obratnega vrstnega reda pa je izračunana napaka višja od pravilne. Čeprav sta operatorja $+$ in $\cdot$ komutativna, pa uporabljeni pristopi tega iz podatkov ne morejo razbrati. Posledično je težko verjeti, da bo pristop izraz „$(1+x)\cdot 9$“ rekonstruiral v „$9\cdot(1+x)$“ in je zato uporaba razdalje urejanja kljub komutativnim operatorjem primerna.

4.2 Rekonstrukcijska napaka

Rekonstrukcijsko napako pristopa na podatkovnih množicah ocenimo s 5-kratnim prečnim preverjanjem. Pristop torej naučimo na štirih izmed petih delov podatkovne množice, na petem pa nato napako ocenimo. To nam da pet rezultatov, enega za vsak del. Uspešnost pristopa torej prikažemo kot povprečje rezultatov, poleg pa dodamo še varianco.

Zaradi stroge strukture izrazov je včasih težko zagotoviti sintaktično pravilnost rekonstruiranih izrazov. Sintaktično nepravilni izrazi so za odkrivanje enačb neuporabni, zato jih pri vrednotenju ne upoštevamo. Za naš pristop (HVAE) to ni problem, saj ta vedno generira sintaktično pravilne izraze. Tudi izrazi, ki jih generira GVAE so v teoriji vedno sintaktično pravilni. GVAE namreč namesto izraza generira zaporedje pravil iz gramatike in pri tem vedno izbira le med dovoljenimi pravili. Problem nastane zaradi omejenega največjega števila pravil, ki včasih ne dovoljuje, da se izraz izpelje do konca. Medtem ko HVAE in GVAE torej z omejitvami zagotovita sintaktično pravilnost izrazov, CVAE takih omejitev nima. Pristop zato generira veliko izrazov, ki so sintaktično nepravilni.

Tabela 1 prikazuje procent sintaktično nepravilnih izrazov na različnih podatkovnih množicah. Vidimo lahko, da HVAE vedno generira izraze, ki so sintaktično pravilni. Ta procent je pri pristopu GVAE nekoliko višji, a v večini primerov pod 0.1%. Veliko večji procent sintaktično nepravilnih izrazov vidimo pri pristopu CVAE. Opazimo lahko, da na pravilnost izrazov pri pristopu CVAE vpliva več stvari. Podatkovne množice, ki vsebujejo tudi funkcije imajo več sintaktično nepravilnih izrazov kot tiste brez njih. Množice z dolgimi izrazi imajo več sintaktično nepravilnih izrazov kot tiste s podobnim številom podatkov in krajšimi izrazi, a vseeno manj kot množice s kratkimi izrazi in malo podatki. Zaradi velikega števila sintaktično nepravilnih izrazov so zato rezultati za CVAE manj zanesljivi in boljši kot bi bili, če bi uporabili tudi sintaktično nepravilne izraze.

Nabor podatkov	HVAE	GVAE	CVAE
AE4-2k	**0.0 (± 0.0)**	0.2 (± 0.0)	33.8 (± 1.1)
Trig4-2k	**0.0 (± 0.0)**	**0.0 (± 0.0)**	48.3 (± 0.6)
AE5-15k	**0.0 (± 0.0)**	< 0.1 (± 0.0)	3.5 (± 0.0)
Trig5-15k	**0.0 (± 0.0)**	< 0.1 (± 0.0)	13.9 (± 0.0)
AE7-20k	**0.0 (± 0.0)**	< 0.1 (± 0.0)	9.9 (± 0.0)
Trig7-20k	**0.0 (± 0.0)**	< 0.1 (± 0.0)	26.3 (± 0.1)

Tabela 1: Procent sintaktično nepravilnih izrazov, ki jih generirajo trije pristopi.

Rekonstrucijska napaka na različnih podatkovnih množicah je prikazana v tabeli 2. Vidimo lahko, da HVAE veliko boljše rekonstruira izraze kot druga dva pristopa. Rezultati za GVAE in CVAE kažejo, da oba pristopa pri rekonstrukciji dosežeta podobno napako. Vidimo tudi, da CVAE bolje dela na podatkovnih množicah brez funkcij, medtem ko GVAE na tistih, ki funkcije vsebujejo. Kot v tabeli 1, se tudi tukaj opazi, da pristopa najbolje rekonstruirata kratke izraze z veliko podatki, najslabše pa kratke izraze z malo podatki.

Nabor podatkov	HVAE	GVAE	CVAE
AE4-2k	**0.027 (± 0.000)**	3.959 (± 0.135)	3.873 (± 0.132)
Trig4-2k	**0.064 (± 0.000)**	3.199 (± 0.068)	3.619 (± 0.045)
AE5-15k	**0.025 (± 0.000)**	2.827 (± 0.280)	1.547 (± 0.466)
Trig5-15k	**0.043 (± 0.000)**	1.489 (± 0.195)	2.086 (± 0.346)
AE7-20k	**0.424 (± 0.001)**	5.201 (± 0.289)	3.654 (± 0.349)
Trig7-20k	**0.424 (± 0.000)**	3.423 (± 0.467)	3.660 (± 0.287)

Tabela 2: Rekonstrukcijska napaka (razdalja urejanj) treh variacijskih samokodirnikov, ocenjena s prečnim preverjanjem.

Vidimo lahko, da pristop GVAE izraze s funkcijami bolje rekonstruira kot tiste brez, HVAE in CVAE pa ravno obratno. Razlog za to je, da funkcije izraz skrajšajo, hkrati pa vplivajo na strukturo izraza. To, da imajo vozlišča s funkcijami le enega naslednika na pristop GVAE ne vpliva, saj so zanj vsa pravila enakovredna. Isto ne velja za pristopa HVAE in CVAE, ki se zaradi spremembe strukture težje naučita rekonstrukcije izraza.

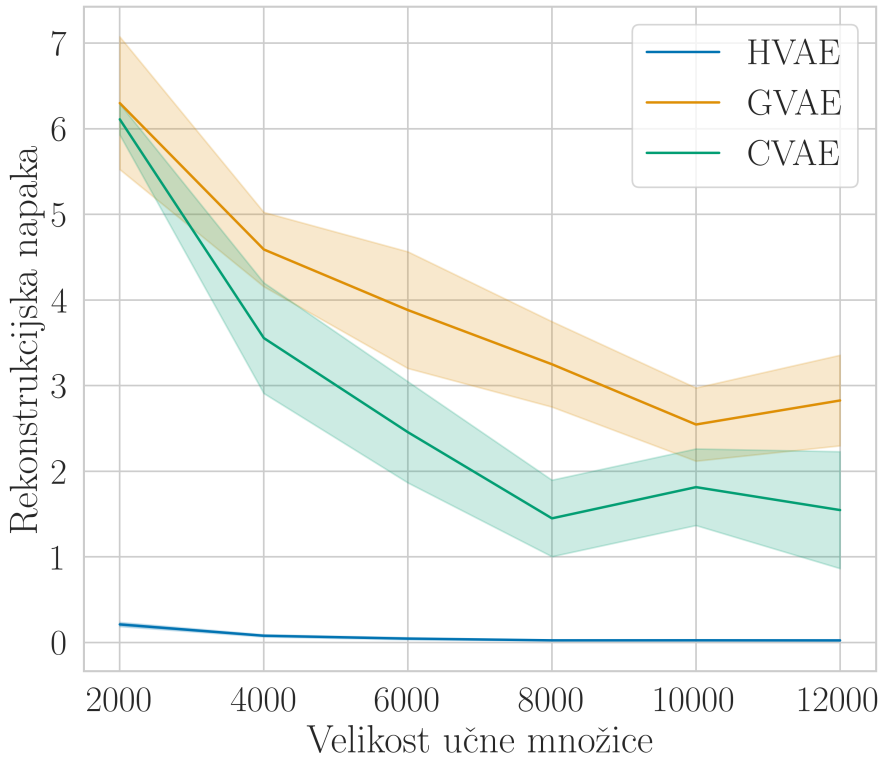
4.2.1 Vpliv velikosti učne množice

Naslednje si bomo ogledali kako na rekonstrukcijo vpliva velikost učne množice. Na področju odkrivanja enačb je namreč velikokrat težko pridobiti veliko množico izrazov, ki so za podan problem primerni.

Uspešnost pristopa pridobimo podobno kot v poglavju 4.2, velikost učne množice pa uravnavamo tako, da podatke v učni množici premešamo in vzamemo prvih $n \in \{2000, 4000, 6000, 8000, 10000, 12000\}$ podatkov. Z nastavljanjem začetnega stanja generatorja naključnih števil pred premešanjem podatkov poskrbimo, da so rezultati ponovljivi in primerljivi med različnimi pristopi. Učno množico pred izbiro učnih podatkov premešamo zato, da se izberejo podatki iz vseh delov učne množice in ne vedno le prvih nekaj.

Uspešnost rekonstrukcije izrazov glede na število podatkov v učni množici je prikazana na sliki 8. Vidimo lahko, da ima naš pristop že pri učni množici velikosti 2000 rekonstrukcijsko napako nižjo od 0.25 in da se ta približa 0, ko je uporabljena učna množica s 6000 ali več izrazi. Naš pristop ima tako šestkrat nižjo napako na učni množici z 2000 izrazi, kot pristop CVAE na učni množici z 12000 izrazi. Kot v poglavju 4.2, tudi tu CVAE bolje rekonstruira izraze kot GVAE na podatkovni množici izrazov brez funkcij.

Zdi se nam, da je glavni razlog za dobro rekonstrukcijo izrazov z modelom HVAE predstavitev izraza z drevesom namesto z nizom. Kot smo omenili v Poglavju 3.1 se v taki predstavitvi podizrazi v izraznem drevesu zakodirajo v enako kodo, ne glede na pozicijo v izrazu. To prostor podizrazov (in posledično izrazov) zelo omeji in s tem olajša učenje in rekonstrukcijo.

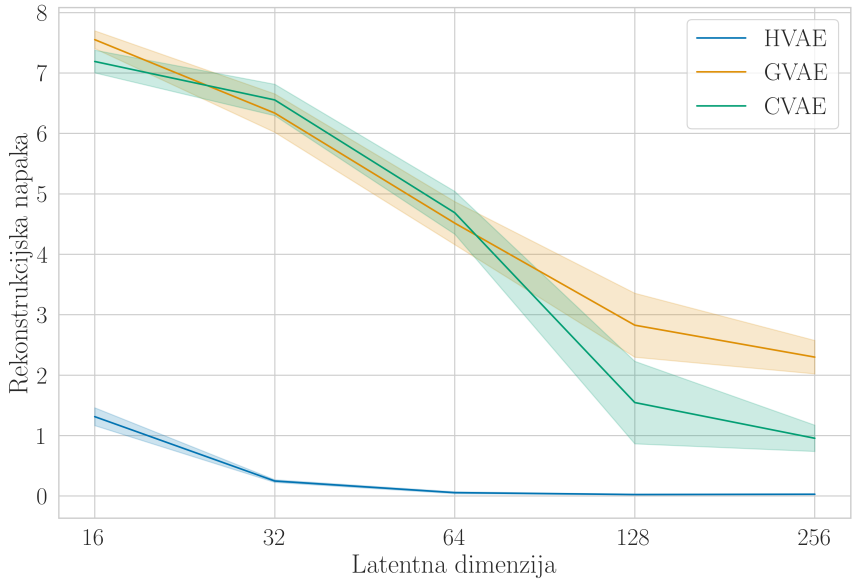


Slika 8: Približki rekonstrukcijske napake (razdalje urejanja), ki jih dobimo s prečnim preverjanjem za različne velikosti učne množice. Uporabljeni izrazi sodijo k podatkovni množici AE5-15k. Barvne črte predstavljajo rekonstrukcijsko napako, pol-prozorni pasovi okoli črt pa varianco.

4.2.2 Vpliv razsežnosti latentnega prostora

Za učinkovito preiskovanje prostora izrazov je pomembno, da je dimenzija latentnega vektorskega prostora čim manjša. Z manjšo dimenzijo pa vektor vsebuje manj informacije in je zato rekonstrukcijska napaka večja. Želimo torej najti velikost latentnega vektorja, ki je čim manjša, a kljub temu enačbe dobro rekonstruira.

Uspešnost rekonstrukcije pri različnih dimenzijah latentnega prostora je prikazana na sliki 9. Uspešnost izračunamo tako kot v poglavju 4.2, a za vsak model prilagodimo velikost latentne in skrite kode. Vidimo lahko, da naš pristop pri $\beta = 16$ (velikost latentne kode) izraze rekonstruira tako dobro ali boljše kot pristopa CVAE in GVAE pri $\beta = 256$. Pristop HVAE ima pri $\beta = 16$ napako približno 1.3, pri $\beta = 32$ se ta napaka spusti pod 0.25, od $\beta = 64$ naprej pa je napaka pod 0.05 in razlika v napaki med različnimi velikostmi zanemarljiva. Vidimo tudi, da pristopa CVAE in GVAE pri manjših velikostih latentnega prostora delata približno enako dobro, pri večjih pa CVAE deluje malo bolje. Pomembno je poudariti tudi, da rezultati niso povsem primerljivi, saj je 90% izrazov, ki jih pristop CVAE generira pri $\beta = 16$ sintaktično nepravilnih. Pri $\beta = 32$ je teh izrazov 63%, pri $\beta = 64$ jih je 10%, od $\beta = 128$ naprej pa število sintaktično nepravilnih izrazov pade pod 4%.



Slika 9: Ocena rekonstrukcijske napake (razdalja urejanja), ki jo dobimo s prečnim preverjanjem za različne velikosti latentnega vektorja. Uporabljeni izrazi sodijo k podatkovni množici AE5-15k. Barvne črte predstavljajo rekonstrukcijsko napako, pol-prozorni pasovi okoli črt pa varianco.

4.3 Linearna interpolacija med izrazi

V poglavju 3.4 smo omenili, da želimo imeti latenten prostor v katerem se točki, ki sta si blizu, preslikata v podoben izraz. To lastnost prostora lahko preverimo z linearno interpolacijo oziroma tako, da naredimo homotopsko (angl. *homotopic*) transformacijo med dvema izrazoma. To naredimo na sledeč način. Naj bosta izraza A in B začetna izraza, torej tista med katerima želimo interpolirati. Izraza pošljemo čez kodirnik in tako dobimo njuni latentni kodi ℓ_A in ℓ_B. S pomočjo ℓ_A, ℓ_B in enačbe $\ell_\alpha = \alpha \cdot \ell_B + (1-\alpha) \cdot \ell_A$, kjer je $\alpha \in \{i/n : i \in \mathbb{N} \wedge i \leq n\}$ nato generiramo latentne kode za nove izraze. Z dekodiranjem dobljenih latentnih kod tako dobimo zaporedje izrazov. V primeru, ko je latenten prostor gladek in so podobni izrazi skupaj zakodirani, bo to zaporedje prikazovalo postopen prehod med izrazoma A in B.

Primeri linearne interpolacije med naključno izbranimi izrazi so za pristop HVAE prikazani v tabeli 3. Vidimo lahko, da je prehod med izrazi postopen, saj se v vsakem koraku spremeni oziroma doda le nekaj simbolov. To je najbolj vidno v primeru ko interpoliramo med izrazom $x^c \cdot \cos c + x$ in $c + \cos \frac{x^c}{c}$. V tem primeru se v prvem koraku x spremeni v x^c, v drugem x v c in x v $\cos x^c$, v tretjem pa se x^c spremeni v $\frac{x^c}{c}$ in $\cos c$ izgine. Čeprav je prehod med vsakim parom izrazov postopen in dokaj intuitiven vidimo tudi, da se generirane latentne kode (ℓ_α) velikokrat dekodirajo v enak izraz. To je posebej vidno na podatkovni množici AE4-2k, kjer so izrazi z $\alpha \in \{0, 0.25\}$ enaki izrazu A in izrazi z $\alpha \in \{0.75, 1\}$ enaki izrazu B.

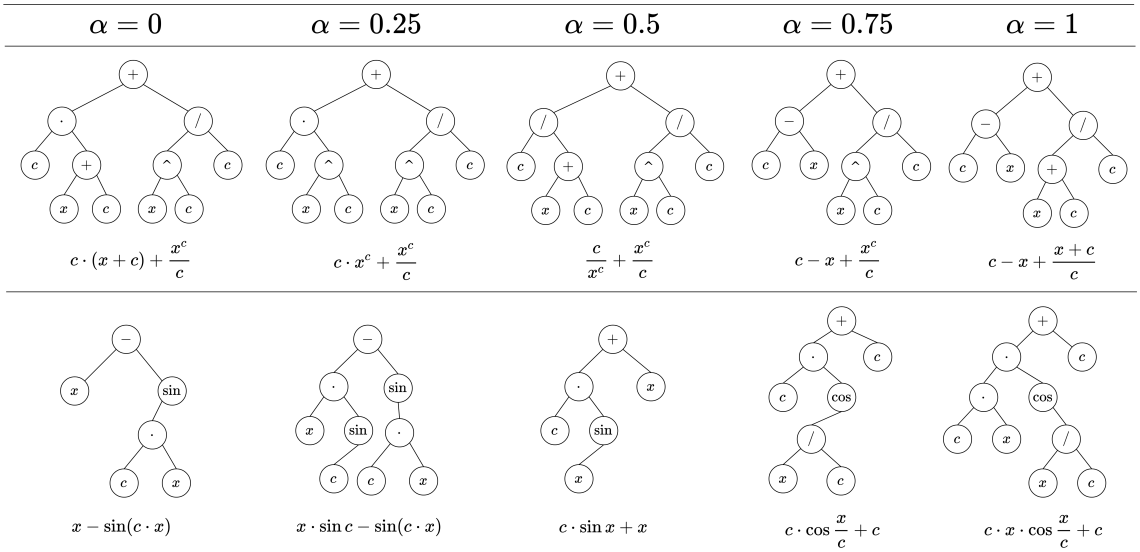
Na sliki 10 je prikazan prehod med izraznim drevesom izraza A in izraza B.

Dataset	α	Primer 1	Primer 2
AE4-2k	Izraz A	$c \cdot (x + c) + \frac{x^c}{c}$	$c - x \cdot c + x$
	$\alpha = 0$	$c \cdot (x + c) + \frac{x^c}{c}$	$c - x \cdot c + x$
	$\alpha = 0.25$	$c \cdot (x + c) + \frac{x^c}{c}$	$c - x \cdot c + x$
	$\alpha = 0.5$	$c \cdot x + \frac{x+c}{c}$	$c - c \cdot x + x$
	$\alpha = 0.75$	$c - x + \frac{x+c}{c}$	$\frac{c+c \cdot x}{c}$
	$\alpha = 1$	$c - x + \frac{x+c}{c}$	$\frac{c+c \cdot x}{c}$
	Izraz B	$c - x + \frac{x+c}{c}$	$\frac{c+c \cdot x}{c}$
AE7-20k	Izraz A	$\frac{c \cdot x^c + x^c}{c} + c$	$\frac{c \cdot x}{c-x} + c - x + c$
	$\alpha = 0$	$\frac{c \cdot x^c + x^c}{c} + c$	$\frac{c \cdot x}{c-x} + c - x + c$
	$\alpha = 0.25$	$\frac{c + x^c}{c} + c$	$c \cdot (c - x) + c - x + c$
	$\alpha = 0.5$	$\frac{c \cdot x^c}{c} + \frac{x}{c}$	$c \cdot x \cdot c - x + c$
	$\alpha = 0.75$	$c - \frac{x^{c^c}}{c}$	$\frac{c \cdot x^c}{c+x} + x$
	$\alpha = 1$	$c - \frac{x^c}{c}$	$\frac{c \cdot x^c}{c+x} + x$
	Izraz B	$c - \frac{x^c}{c}$	$\frac{c \cdot x^c}{c+x} + x$
Trig5-15k	Izraz A	$c + \sin c + \frac{\sin x}{x} - \frac{x}{c}$	$x^c \cdot \cos c + x$
	$\alpha = 0$	$c + \sin c + \frac{\sin x}{x} - \frac{x}{c}$	$x^c \cdot \cos c + x$
	$\alpha = 0.25$	$c + c + \frac{\sin x}{c} - \frac{x}{c}$	$x \cdot \cos c + x$
	$\alpha = 0.5$	$x + \frac{x}{c} - \frac{x}{c}$	$c \cdot \cos c + \cos x^c$
	$\alpha = 0.75$	$x + c - x \cdot \sin c$	$c + \cos \frac{x^c}{c}$
	$\alpha = 1$	$x^c - x \cdot \sin c$	$c + \cos \frac{x^c}{c}$
	Izraz B	$x^c - x \cdot \sin c$	$c + \cos \frac{x^c}{c}$

Tabela 3: Primeri linearne interpolacije med dvema naključno izbranima izrazoma na različnih podatkovnih množicah. Prva in zadnja vrstica v vsakem primeru predstavljata začeten izraz, vrstice z $\alpha = x$ pa dekodiran izraz, ki pripada generiranemu vektorju ℓ_x.

Tu lahko še bolje opazimo postopen prehod med strukturo izraznih dreves in posledično izrazoma. V vsakem koraku se zamenja le nekaj simbolov, struktura izraznega drevesa pa se le malo spremeni.

Nazadnje v tabeli 4 prikažemo kako različni pristopi interpolirajo med dvema izrazoma. Vidimo, da je prehod med izrazoma pri uporabi pristopa HVAE postopen in da so v večini primerov koraki, ki jih naredi samokodirnik pričakovani. Tudi pristop GVAE je pri prehodu postopen, a uporabljeni koraki niso vedno taki, kot bi pričakovali (primer 1). Bolj pričakovani so koraki, ki jih naredi CVAE. Velika slabost CVAE-ja pa je, da so nekateri izmed dekodiranih izrazov sintaktično nepravilni in posledično neuporabni pri odkrivanju enačb.



Slika 10: Prikaz spreminjanja izraznega drevesa med linearno interpolacijo. Zaradi točne rekonstrukcije izrazov A in B, so drevesa le-teh izpuščena.

Primer	α	HVAE	GVAE	CVAE
Primer 1	Izraz A	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin x}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$
	$\alpha = 0$	$c \cdot x + \frac{c}{x} + \frac{x}{\sin x}$	$c \cdot \cos c + \frac{c}{x} + \frac{x}{\sin c}$	$c \cdot \cos c + \frac{x}{x^x} \cdot \sin x$
	$\alpha = 0.25$	$c \cdot \sin c + \frac{c}{x} + x$	$c + \frac{c}{\cos c} + c \cdot x$	$\frac{c}{\cos c} + c - \frac{x}{c} \cdot \sin x$
	$\alpha = 0.5$	$\frac{c \cdot \sin c}{c} + x$	$c + \frac{x}{c^{\cos c}} \cdot x$	$c - \cos c - \frac{x}{x} \cdot \sin c$
	$\alpha = 0.75$	$\frac{\sin(x - \sin c)}{x}$	$x + c \cdot \frac{c}{x} - x$	$\cos(x(c) - c \cdot$
	$\alpha = 1$	$\frac{\sin(x - c)}{x}$	$\frac{\sin(x - c)}{x}$	$\frac{\sin(x - c)}{x}$
	Izraz B	$\frac{\sin(x - c)}{x}$	$\frac{\sin(x - c)}{x}$	$\frac{\sin(x - c)}{x}$
Primer 2	Izraz A	$x - \sin(c \cdot x)$	$x - \sin(c \cdot x)$	$x - \sin(c \cdot x)$
	$\alpha = 0$	$x - \sin(c \cdot x)$	$\frac{x}{\sin(c \cdot x)}$	$x - \sin(c \cdot x)$
	$\alpha = 0.25$	$x \cdot \sin c - \sin(c \cdot x)$	$\frac{x}{\sin(c \cdot x)}$	$x - \cos(c \cdot x)$
	$\alpha = 0.5$	$c \cdot \sin x + x$	$c + \cos c$	$\frac{c}{\cos} - \cos \cdot c) - c$
	$\alpha = 0.75$	$c \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \frac{\cos(\frac{x}{c})}{c}$	$c \cdot x \cdot \frac{\cos(\frac{c}{x})}{c}$
	$\alpha = 1$	$c \cdot x \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$
	Izraz B	$c \cdot x \cdot \cos \frac{x}{c} + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$	$c \cdot x \cdot \cos(\frac{x}{c}) + c$

Tabela 4: Primeri linearne interpolacije med izrazi za različne variacijske samokodirnike. Rdeče obarvani izrazi so sintaktično nepravilni.

4.4 Odkrivanje enačb

Prostor možnih izrazov je velik, zato je v večini primerov sistematično preiskovanje v njem neobvladljivo. Bayesova optimizacija v evklidskem prostoru omogoča, da preiskovanje usmerimo v dele prostora, v katerem so izrazi bolj 'perspektivni'. V prejšnjem podpoglavju smo pokazali, da pristop HVAE podobne izraze zakodira blizu v latentnem prostoru. To nam omogoča preiskovanje latentnega prostora z uporabo Bayesove optimizacije ob predpostavki, da podobni izrazi dosežejo približno enako napako na določeni množici podatkov.

Pristop preizkusimo na štirih enačbah različne kompleksnosti iz Feynmanove podatkovne množice za simbolno regresijo [44]. Za vsako izmed izbranih enačb naključno izberemo $10,000$ izmed milijona simuliranih meritev in postopek iskanja ponovimo trikrat. V vsakem poskusu Bayesovo optimizacijo inicializiramo s 512-imi izrazi in nato opravimo 64 iteracij, v katerih vzorčimo po 16 izrazov. Naš pristop primerjamo s ProGED-om, ki izraze generira z naključnim (Monte Carlo) vzorčenjem. V poskusih s ProGED-om iz verjetnostne gramatike vzorčimo $512+16\cdot 64 = 1536$ izrazov. Verjetnostna gramatika, ki jo uporabimo je ekvivalentna tisti, ki jo uporabimo za generiranje učnih primerov za naš pristop.

Primerjava s pristopom ProGED je prikazana v tabeli 5. Vidimo lahko, da preprostejše izraze skoraj vedno odkrijemo, medtem ko kompleksnejšega (zadnjega) ne odkrijemo v nobenem poskusu. Vidimo lahko, da naš pristop iskane izraze najde v več poskusih. Ko oba pristopa iskanega izraza ne najdeta, pa naš pristop (v vseh poskusih) najde izraz, ki se danim podatkom bolje prilega.

	HVAE		ProGED [6]	
Iskan izraz	Najboljši MSE	Uspešna rekonstrukcija	Najboljši MSE	Uspešna rekonstrukcija
$2 \cdot \pi \cdot \frac{x_1}{x_2 \cdot x_3}$	0.000	3	0.000	2
$x_1 \cdot x_2 \cdot \sin x_3$	0.000	2	0.000	1
$\frac{x_1}{2 \cdot (1+x_2)}$	0.000	3	0.001	0
$2 \cdot x_1 \cdot (1 - \cos(x_2 \cdot x_3))$	8.358	0	19.886	0

Tabela 5: Primerjava rekonstrukcije izrazov s pristopom HVAE in ProGED-om.

Tabela 6 prikazuje izraze, ki jih naš pristop najde s pomočjo Bayesove optimizacije. Vidimo lahko, da naš pristop včasih poleg točnega izraza odkrije tudi njegove izpeljave. To je najbolj opazno pri odkrivanju tretje enačbe v tabeli, kjer Bayesova optimizacija najde izraz $-0.5 \cdot \frac{x_1}{-x_2-1}$. Zanimivo je tudi, da naš pristop pri neuspešnem poskusu odkrivanja drugega izraza najde izraz $x_2 \cdot \sin x_3 \cdot x_1 - x_1$, ki se od iskanega $x_2 \cdot \sin x_3 \cdot x_1$ razlikuje le za en člen.

	Najboljši poskus		Najslabši poskus	
Iskan izraz	Rekonstruiran izraz	MSE	Rekonstruiran izraz	MSE
$2 \cdot \pi \cdot \frac{x_1}{x_2 \cdot x_3}$	$6.28 \cdot \frac{x_1}{x_2 \cdot x_3}$	0.000	$\frac{6.28}{x_2} \cdot \frac{x_1}{x_3}$	0.000
$x_1 \cdot x_2 \cdot \sin x_3$	$x_2 \cdot x_1 \cdot \sin x_3$	0.000	$x_2 \cdot \sin x_3 \cdot x_1 - x_1$	2.393
$\frac{x_1}{2 \cdot (1+x_2)}$	$-0.5 \cdot \frac{x_1}{-x_2-1}$	0.000	$\frac{0.5 \cdot x_1^1}{(x_2+1)^1}$	0.000
$2 \cdot x_1 \cdot (1 - \cos(x_2 \cdot x_3))$	$1.1 \cdot x_1^{1.3} + \frac{9}{x_3 \cdot (x_2+0.2)}$	8.358	$2 \cdot x_1 + \frac{x_1}{x_3} + \sin(x_2 + 26)$	19.569

Tabela 6: Rezultati odkrivanja enačb z Bayesovo optimizacijo nad latentnim prostorom pristopa HVAE.

5 Zaključek

V delu predstavimo pristop za generiranje algebrajskih izrazov, ki temelji na variacijskih samokodirnikih. Uporabljamo predstavitev izrazov z dvojiškimi drevesi.

Ker naš generator tvori le pravilna dvojiška drevesa, je sintaktična pravilnost dobljenih izrazov zagotovljena, kar ni samoumevno pri alternativnih pristopih. Poleg tega z binarnim drevesom implicitno povemo, da so podizrazi med seboj neodvisni. Model je zato bolj učinkovit od alternativnih pristopov, ob tem pa ne uporablja dodatnih abstrakcij (gramatik in izpeljav) in z njimi povezanim nepotrebnim računskim bremenom. Učinkovitost modela pokažemo v poglavju 4, kjer ga primerjalno vrednotimo z alternativnimi generatorji, ki temeljijo na variacijskih samokodirnikih.

Rezultati primerjalnega vrednotenja pokažejo, da naš pristop v primerjavi z GVAE in CVAE izboljša rekonstrukcijo algebrajskih izrazov tudi ob uporabi manjše podatkovne množice in latentnega vektorskega prostora nižje dimenzije. To pokažemo v poglavju 4.2.1, kjer naš pristop rekonstruira izraze več kot šestkrat bolje kljub šestkrat manjši učni množici in v poglavju 4.2.2, kjer naš pristop izraze podobno dobro rekonstruira kljub šestnajstkrat nižji dimenziji latentnega prostora. Slednji rezultat je še posebej pomemben v kontekstu uporabe Bayesove optimizacije, ki je zelo občutljiva na (visoko) dimenzionalnost optimizacijskega prostora. Poleg tega z linearno interpolacijo v poglavju 4.3 in Bayesovo optimizacijo v poglavju 4.4 pokažemo, da latentni prostor zagotavlja gladke prehode med podobnimi algebrajskimi izrazi in je uporaben za odkrivanje enačb.

V prihodnjem delu želimo naš pristop preizkusiti na nalogah rekonstrukcije vseh enačb iz Feynman-ove [44] in Nguyen-ove [45] množice primerjalnih testov za simbolno regresijo. S tem bi odkrili potencial pristopa za simbolno regresijo in ga postavili v kontekst z drugimi algoritmi za odkrivanje enačb in simbolno regresijo, tudi tistimi, ki ne temeljijo na variacijskih samokodirnikih. Poleg tega želimo izboljšati implementacijo pristopa, saj zaradi razlik v strukturi dreves trenutno ne omogoča združevanje dreves med učenjem. Združevanje trenutno le simuliramo tako, da naenkrat čez model pošljemo eno drevo, nato pa kot napako za skupino vzamemo povprečje posameznih napak. Boljša implementacija bi pristop še dodatno pohitrila in s tem povečala njegovo uporabnost. Nazadnje, v trenutnem pristopu ne upoštevamo vsega predznanja, ki ga o izrazih premoremo, na primer komutativnosti operatorjev za seštevanje in množenje [1]. Z upoštevanjem podobnega predznanja želimo v prihodnje model in kvaliteto latentnega prostora še izboljšati.

Literatura

[1] M. Allamanis in dr., *Learning continuous semantic representations of symbolic expressions*, v: Proceedings of the 34th International Conference on Machine Learning - Volume 70, JMLR.org, 2017, str. 80–88.

[2] D. Alvarez-Melis in T. Jaakkola, *Tree-structured decoding with doubly-recurrent neural networks*, v: ICLR, 2017.

[3] H. Bahuleyan, *Natural language generation with neural variational models*, University of Waterloo, Canada (2018), `https://arxiv.org/pdf/1808.09012.pdf`.

[4] M. Balandat in dr., *BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization*, v: Advances in Neural Information Processing Systems 33, 2020.

[5] D. M. Blei, A. Kucukelbir in J. D. McAuliffe, *Variational inference: A review for statisticians*, Journal of the American Statistical Association **112**(518) (2017) 859–877, doi: 10.1080/01621459.2017.1285773.

[6] J. Brence, L. Todorovski in S. Džeroski, *Probabilistic grammars for equation discovery*, Knowledge-Based Systems **224** (2021) 107077.

[7] W. Bridewell in dr., *Inductive process modeling*, Machine Learning **71**(1) (2008) 1–32, doi: 10.1007/s10994-007-5042-6.

[8] T. Brown in dr., *Language models are few-shot learners*, v: Advances in Neural Information Processing Systems (ur. H. Larochelle in dr.), **33**, Curran Associates, Inc., 2020, str. 1877–1901.

[9] S. L. Brunton, J. L. Proctor in J. N. Kutz, *Discovering governing equations from data by sparse identification of nonlinear dynamical systems*, Proceedings of the National Academy of Sciences **113**(15) (2016) 3932–3937.

[10] K. Cho in dr., *Learning phrase representations using RNN encoder–decoder for statistical machine translation*, v: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Doha, Qatar, 2014, str. 1724–1734, doi: 10.3115/v1/D14-1179.

[11] H. Dai in dr., *Syntax-directed variational autoencoder for structured data*, 2018.

[12] E. Dijkstra, *Algol 60 translation : An Algol 60 translator for the X1 and making a translator for Algol 60*, 1961.

[13] S. Dzeroski in L. Todorovski, *Discovering dynamics: From inductive logic programming to machine discovery*, Journal of Intelligent Information Systems **4**(1) (1995) 89–108, doi: 10.1007/BF00962824.

[14] S. Džeroski in L. Todorovski, *Equation discovery for systems biology: finding the structure and dynamics of biological networks from time course data*, Current Opinion in Biotechnology **19**(4) (2008) 360–368, doi: https://doi.org/10.1016/j.copbio.2008.07.002, protein technologies / Systems biology.

[15] B. C. Falkenhainer in R. S. Michalski, *Integrating quantitative and qualitative discovery: The abacus system*, Mach. Learn. **1**(4) (1986) 367–401, doi: 10.1023/A:1022866732136.

[16] W. Fedus, I. Goodfellow in A. M. Dai, *MaskGAN: Better text generation via filling in the _*, v: International Conference on Learning Representations, 2018.

[17] I. Goodfellow, Y. Bengio in A. Courville, *Deep Learning*, MIT Press, 2016, `http://www.deeplearningbook.org`.

[18] R. Guimerà in dr., *A bayesian machine scientist to aid in the solution of challenging scientific problems*, Science Advances **6**(5) (2020) eaav6971, doi: 10.1126/sciadv.aav6971.

[19] R. Gómez-Bombarelli in dr., *Automatic chemical design using a data-driven continuous representation of molecules*, ACS Central Science **4**(2) (2018) 268 – 276, doi: 10.1021/acscentsci.7b00572.

[20] G. Hinton in dr., *Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups*, IEEE Signal Processing Magazine **29**(6) (2012) 82–97, doi: 10.1109/MSP.2012.2205597.

[21] G. E. Hinton in R. R. Salakhutdinov, *Reducing the dimensionality of data with neural networks*, Science **313**(5786) (2006) 504–507, doi: 10.1126/science.1127647.

[22] G. E. Hinton in R. Zemel, *Autoencoders, minimum description length and helmholtz free energy*, v: Advances in Neural Information Processing Systems (ur. J. Cowan, G. Tesauro in J. Alspector), **6**, Morgan-Kaufmann, 1993.

[23] S. Hochreiter in J. Schmidhuber, *Long short-term memory*, Neural Comput. **9**(8) (1997) 1735–1780, doi: 10.1162/neco.1997.9.8.1735.

[24] W. Jin, R. Barzilay in T. Jaakkola, *Junction tree variational autoencoder for molecular graph generation*, v: Artificial Intelligence in Drug Discovery, The Royal Society of Chemistry, 2021, str. 228–249, doi: 10.1039/9781788016841-00228.

[25] D. P. Kingma in M. Welling, *An introduction to variational autoencoders*, Foundations and Trends® in Machine Learning **12**(4) (2019) 307–392, doi: 10.1561/2200000056.

[26] M. M. Kokar, *Determining arguments of invariant functional descriptions*, Machine Learning **1**(4) (1986) 403–422, doi: 10.1023/A:1022818816206.

[27] J. R. Koza, *Genetic programming as a means for programming computers by natural selection*, Statistics and Computing **4**(2) (1994) 87–112, doi: 10.1007/BF00175355.

[28] S. Kullback in R. A. Leibler, *On Information and Sufficiency*, The Annals of Mathematical Statistics **22**(1) (1951) 79 – 86, doi: 10.1214/aoms/1177729694.

[29] M. J. Kusner, B. Paige in J. M. Hernández-Lobato, *Grammar variational autoencoder*, v: Proceedings of the 34th International Conference on Machine Learning - Volume 70, ICML'17, JMLR.org, 2017, str. 1945–1954.

[30] C. Lassner, G. Pons-Moll in P. V. Gehler, *A generative model for people in clothing*, v: Proceedings of the IEEE International Conference on Computer Vision, 2017.

[31] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, Soviet physics. Doklady **10** (1965) 707–710.

[32] A. Meurer in dr., *SymPy: symbolic computing in Python*, PeerJ Computer Science **3** (2017) e103, doi: 10.7717/peerj-cs.103.

[33] C. E. Rasmussen in C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*, The MIT Press, 2005.

[34] S. Ruder, *An overview of gradient descent optimization algorithms.*, 2016.

[35] C. Schaffer, *Bivariate scientific function finding in a sampled, real-data testbed*, Machine Learning **12**(1) (1993) 167–183, doi: 10.1023/A:1022879602886.

[36] M. Schmidt in H. Lipson, *Distilling free-form natural laws from experimental data*, Science **324**(5923) (2009) 81–85, doi: 10.1126/science.1165893.

[37] B. Shahriari in dr., *Taking the human out of the loop: A review of bayesian optimization*, Proceedings of the IEEE **104**(1) (2016) 148–175.

[38] J. Snoek, H. Larochelle in R. P. Adams, *Practical bayesian optimization of machine learning algorithms*, v: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2, NIPS'12, Curran Associates Inc., Red Hook, NY, USA, 2012, str. 2951–2959.

[39] I. Sobol', *On the distribution of points in a cube and the approximate evaluation of integrals*, USSR Computational Mathematics and Mathematical Physics **7**(4) (1967) 86–112, doi: https://doi.org/10.1016/0041-5553(67)90144-9.

[40] R. Storn in K. Price, *Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces*, Journal of Global Optimization **11**(4) (1997) 341–359, doi: 10.1023/A:1008202821328.

[41] I. Sutskever, O. Vinyals in Q. V. Le, *Sequence to sequence learning with neural networks*, v: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14, MIT Press, Cambridge, MA, USA, 2014, str. 3104–3112.

[42] K. S. Tai, R. Socher in C. D. Manning, *Improved semantic representations from tree-structured long short-term memory networks*, v: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), Association for Computational Linguistics, Beijing, China, 2015, str. 1556–1566, doi: 10.3115/v1/P15-1150.

[43] L. Todorovski in S. Dzeroski, *Declarative bias in equation discovery*, v: Proceedings of the Fourteenth International Conference on Machine Learning, Morgan Kaufmann, 1997, str. 376–384.

[44] S.-M. Udrescu in M. Tegmark, *Ai feynman: A physics-inspired method for symbolic regression*, Science Advances **6**(16) (2020) eaay2631, doi: 10.1126/sciadv.aay2631.

[45] N. Q. Uy in dr., *Semantically-based crossover in genetic programming: application to real-valued symbolic regression*, Genetic Programming and Evolvable Machines **12**(2) (2011) 91–119, doi: 10.1007/s10710-010-9121-2.

[46] P. Vincent in dr., *Extracting and composing robust features with denoising autoencoders*, v: Proceedings of the 25th International Conference on Machine Learning, ICML '08, Association for Computing Machinery, New York, NY, USA, 2008, str. 1096–1103, doi: 10.1145/1390156.1390294.

[47] T. Washio in H. Motoda, *Discovering admissible models of complex systems based on scale-types and identity constraints*, v: Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2, IJCAI'97, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997, str. 810–817.

[48] D. Whitley, *A genetic algorithm tutorial*, Statistics and Computing **4**(2) (1994) 65–85, doi: 10.1007/BF00175354.

[49] C. Yang in dr., *High-resolution image inpainting using multi-scale neural patch synthesis*, v: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.

[50] Z. Yang in dr., *Semi-supervised QA with generative domain-adaptive nets*, v: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Vancouver, Canada, 2017, str. 1040–1050, doi: 10.18653/v1/P17-1096.

[51] Z. Yi in dr., *Dualgan: Unsupervised dual learning for image-to-image translation*, v: 2017 IEEE International Conference on Computer Vision (ICCV), 2017, str. 2868–2876, doi: 10.1109/ICCV.2017.310.

[52] A. Zhang in dr., *Dive into deep learning*, arXiv preprint arXiv:2106.11342 (2021) 347–392.