Određivanje stupnjeva slobode pomoću dubokog učenja

Pavlović, Matej

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

Matej Pavlović

Određivanje stupnjeva slobode pomoću dubokog učenja

Diplomski rad

Zagreb, 2020.

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

INTEGRIRANI PREDDIPLOMSKI I DIPLOMSKI SVEUČILIŠNI
STUDIJ FIZIKA; SMJER ISTRAŽIVAČKI

Matej Pavlović

Diplomski rad

Određivanje stupnjeva slobode pomoću dubokog učenja

izv. prof. dr. sc. Davor Horvatić

Ocjena diplomskog rada: _____

Povjerenstvo: 1. _____

2. _____

3. _____

Datum polaganja: _____

Zagreb, 2020.

Zahvaljujem se mentoru, izv. prof. dr. sc. Davoru Horvatiću na savjetima i ponajviše na slobodi prilikom odabira teme diplomskog rada i načina njezine realizacije. Također se zahvaljujem svima koji su mi bili potpora tijekom studija.

Sažetak

U ovom radu predstavljena je metodologija za određivanje kritične temperature fizikalnih sustava. Metoda se zasniva na korištenju neuronskih mreža čija se arhitektura ponaša poput renormalizacijske grupe (RG). Metodologija je primijenjena na 2D Isingovom modelu za tri različite rešetke: kvadratnu, trokutastu i heksagonalnu. Naučena su tri generativna modela koji imaju ulogu RG-a, a to su: ograničen Boltzmannov stroj, varijacijski autoenkoder i generativne suparničke mreže. Iterativnom primjenom generativnih modela generiramo sustave u blizini kritične temperature. Temperature sustava određene su pomoću naučene neuronske mreže. Također je primijenjena i metoda bez generatora koja se zasniva na iterativnoj primjeni normalizacije skrivenih vektora mreže. Metoda bez generatora i ograničen Boltzmannov stroj daju dobre i stabilne rezultate za sve tri rešetke, dok varijacijskom autoenkoderu i generativnim suparničkim mrežama točnost varira od rešetke do rešetke.

Ključne riječi: Isingov model, kritična temperatura, strojno učenje, ograničen Boltzmannov stroj, varijacijski autoenkoder, generativne suparničke mreže

Determining degrees of freedom with deep learning

Abstract

This thesis presents a methodology for determining the critical temperature of a physical system. The method is based on neural networks that behave like a renormalization group (RG). The methodology was applied to the 2D Ising model on three different lattices: square, triangular, and hexagonal. Three generative models that behave like the RG have been trained, namely: a restricted Boltzmann machine, a variational autoencoder, and generative adversarial networks. By iteratively applying generative models to the data, we obtained systems close to the critical temperature. System temperatures were measured using a neural network that was trained for this purpose. A generator-free method based on the iterative application of normalization on the network's hidden vectors is also applied. The generator-free method and the limited Boltzmann machine give good and stable results for all three lattices, while the variation of the autoencoder and generative adversarial networks vary from lattice to lattice.

Keywords: Ising model, critical temperature, machine learning, restricted Boltzman machine, variational autoencoder, generative adversarial networks

Sadržaj

1 Uvod **1**

2 Fazni prijelazi u spinskim sustavima **3**

2.1 Korelacijska dužina 4

2.2 Renormalizacijska grupa 6

2.3 Isingov model 8

3 Neuronske mreže **10**

3.1 Od neurona do mreže 11

3.2 Učenje mreže 13

4 Generativni modeli **15**

4.1 Ograničeni Boltzmannov stroj 15

4.2 Varijacijski autoenkoder 17

4.3 Generativne suparničke mreže 19

4.4 Tok renormalizacijske grupe i generativni modeli 20

5 Implementacija problema na Isingovom modelu **22**

5.1 Generiranje podataka 22

5.2 Modeli 23

5.2.1 Temperaturni model 23

5.2.2 Ograničen Boltzmannov stroj 24

5.2.3 Varijacijski autoenkoder 25

5.2.4 Generativna suparnička mreža 25

5.3 Tok renormalizacije grupe 27

5.4 Tok bez generatora 28

6 Zaključak **30**

Dodaci **31**

A Kod **31**

A.1 Kod za generiranje podataka 31

A.2 Kod s modelima 35

A.3 Kod s pomoćnim funkcijama 38
A.4 Kod za učenje i generiranje 45

Literatura **52**

1 Uvod

Strojno učenje je područje istraživanja koje u zadnjih deset godina doživljava velik rast i napredak. U svom radu [1], Andrew Ng uveo je učenje neuronskih mreža na grafičkim karticama (engl. *graphics processing unit* (GPU)). Učenje na GPU-ima dalo je ubrzanja od barem reda veličine što je omogućilo korištenje većih modela na više podataka, pa se samim time i točnost modela znatno povećala.

Strojno učenje, a posebno duboko učenje, vuče korijenje iz teorijske fizike i kemije. Metode strojnog učenja se u fizici najviše koriste kod čestične fizike, astrofizike, fizike čvrstog stanja i ubrzavanja numeričkih simulacija. Interpretacija neuronskih mreža još uvijek je predmet istraživanja, no mogu se povezati s konceptom renormalizacijske grupe u fizici. Renormalizacijske grupe su način konstrukcije značajki na makroskopskoj skali polazeći od mikroskopske skale. Uzme li se za primjer klasifikacija slika, mreža za donošenje odluke kreće od promatranja svakog piksela pojedinačno te, kako ide u dublje slojeve, gradi značajke koje opisuju velike dijelove slike.

U ovom radu neuronske mreže korištene su za pronalaženje kritične temperature sustava. Za fizikalni sustav izabran je Isingov model primijenjen na tri različite rešetke (kvadratnu, trokutastu i heksagonalnu), a u obzir se uzimaju interakcije samo s najbližim susjedima. Trenira se generator koji uči distribuciju podataka u svrhu generiranja istih, te model za temperaturu pomoću kojeg se mjere temperature generiranih sustava. Generator se ponaša kao iterativna primjena renormalizacijske grupe u svrhu konvergiranja u kritičnu temperaturu. U svrhu generatora korišteni su ograničen Boltzmannov stroj, varijacijski autoenkoder i generativne suparničke mreže.

Također, napravljen je i pristup bez generatora koji se sveo na iterativnu normalizaciju skrivenih vektora modela za temperaturu. Ove metoda daje najbolje rezultate i zahtijeva učenje samo jednog modela s normalizacijskim slojevima.

U drugom poglavlju objašnjeni su osnovni koncepti faznih prijelaza u spinskim sustavima potrebne za razumijevanje rada. Pokazana je uloga korelacijske dužine i renormalizacijke grupe u faznim prijelazima, te objašnjene su osnove Isingovog modela. U trećem poglavlju uveden je koncept neuronskih mreža. Detaljnije su objašnjeni osnovni gradivni blokovi mreže i postupak učenja mreže na podacima. U četvrtom poglavlju uvedeni su generativne modele koji su se koristili u radi. Nakon detaljnijeg objašnjenja svakog modela posebno, pokazana je sličnost između renormalizacijske grupe i genera-

tivnih modela. U petom poglavlju dana je implementacija problema na Isingovom modelu za tri različite rešetke. Također, analizirani su rezultati za pristup s generatorom i pristup bez generatora.

2 Fazni prijelazi u spinskim sustavima

Fazni prijelaz je svaki događaj nakon kojeg sustav mijenja svoje ponašanje. Promjena ponašanja može se manifestirati na različite načine, od promjene agregatnog stanja pa do promjene u uređenju atoma sustava. To za posljedicu ima promjenu vrijednosti nekog svojstva sustava (magnetizacija, vodljivost, itd.). Događaj koji uzrokuje fazni prijelaz najčešće je promjena temperature iznad neke točke, no postoje i drugi mehanizmi poput promjene tlaka. Kada se razmatra fazni prijelaz na sustavu spinova, takvi sustavi fazni prijelaz manifestiraju preko uređenja sustava.

Feromagneti su sustavi koji posjeduju magnetizaciju čak i kada primijenjeno magnetsko polje iščezava (spontana magnetizacija). Takvo ponašanje sugerira da se elektron i magnetski momenti nalaze u nekoj vrsti uređenja. U slučaju paramagneta s koncentracijom od N iona spina S, interno međudjelovanje poravnava spinove kako bi svi bili paralelni što u konačnici rezultira ukupnom magnetizacijom različitom od nule. Interakcija se može modelirati poljem izmjene (B_E) koje mora savladati termalne fluktuacije. Ako temperatura postane prevelika, termalne fluktuacije nadjačaju polje izmjene te se spinska uređenost uništava.

Teorija srednjeg polja [2] pretpostavlja da na svaki atom djeluje polje izmjene koje je proporcionalno magnetizaciji. Tada se polje izmjene može izraziti kao:

$$B_E = \lambda M, \tag{2.1}$$

gdje je λ konstanta koja ne ovisi o temperaturi. Jednadžba pokazuje da na svaki spin djeluje prosječna magnetizacija svih ostalih spinova.

Curiejeva temperatura T_C dijeli uređeno feromagnetsko stanje i neuređeno paramagnetsko stanje. To znači da na temperaturi T_C materijali gube svojstvo spontane magnetizacije. Za $T < T_C$ materijal je u uređenoj feromagentskoj fazi, dok je za $T > T_C$ u neuređenoj paramagnetskoj fazi.

Primijenimo li magnetsko polje B_a na materijal koji se nalazi u paramagnetskoj fazi, dolazi do određene magnetizacije te do generiranja polja B_E. Magnetizaciju M možemo izraziti kao:

$$\mu_0 M = \chi_p (B_a + B_E), \tag{2.2}$$

gdje je χ_p paramagnetska susceptibilnost. Nadalje, paramagnetska susceptibilnost dana

je Curiejevim zakonom:

$$\chi_p = C/T, \tag{2.3}$$

gdje je C Curiejeva konstanta. Uvrštavanjem jednadžbe 2.2 u 2.1 te iskorištavanjem relacije 2.3 slijedi:

$$\mu_0 MT = C(B_a + \lambda M). \tag{2.4}$$

Iz definicije susceptibilnosti:

$$\chi = \frac{\mu_0 M}{B}, \tag{2.5}$$

naposljetku se dobiva:

$$\chi = \frac{M}{B_a} = \frac{C}{T - C\lambda}. \tag{2.6}$$

Gornja jednadžba divergira na temperaturi $T = C\lambda$. Ta točka naziva se Curiejeva točka i predstavlja granicu između feromagnetskog i paramagnetskog stanja, odnosno temperaturu faznog prijelaza.

Polje izmjene aproksimacija je kvantno-mehaničke interakcije čestica. Egzaktna energija interakcija U dana je Heisenbergovim modelom:

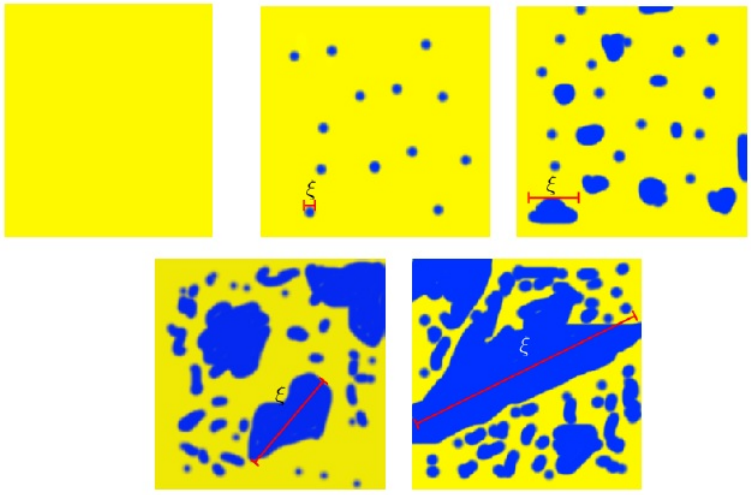
$$U = -2JS_1 \cdot S_2, \tag{2.7}$$

gdje je J interakcijska konstanta koja ovisi o preklopu dviju valnih funkcija, a S_1 i S_2 spinovi pripadnih atoma. U feromagentskoj fazi atomi teže poravnanju sa susjedima pa prema tome postoji određena korelacija između svih atoma.

2.1 Korelacijska dužina

Teorija srednjeg polja radi vrlo veliku pretpostavku, a to je da zanemaruje fluktuacije. Statistička fizika temelji se na proučavanju fluktuacija te ih je potrebno uzeti u obzir prilikom računa. Korelacijska dužina ξ korisna je veličina koja govori koliki je doseg interakcije. Na primjer, ako se preokrene jedan spin, ta akcija utječe na distribuciju spinova do udaljenosti ξ. Vrijednost korelacijske dužine pokazuje koliki je radijus djelovanja termalnih fluktuacija. Na visokim temperaturama, fluktuacije su jake te unište korelaciju čime je vrijednost korelacijske dužine mala. Na niskim temperaturama, gotovo pa nema termalnih fluktuacija pa je ξ također mala. U srednjem režimu temperatura, korelacijska dužina može poprimiti ogromne vrijednosti. Kako je fazni prijelaz okarakteriziran

velikim fluktuacijama, time ξ čini dosta važnu veličinu u blizini kritične točke (slika 2.1).



Slika 2.1: Prikaz rasta korelacijske dužine u blizini faznog prijelaza. Domene nisu statičke nego se mijenjaju u vremenu [4].

Kako bi se lakše kvantificirali rezultati, definira se operator [3] koji određuje nalazi li se sustav u uređenoj ili neuređenoj fazi. Promatranjem čestica i i j može se uvesti mjera $r = |i - j|$ koja predstavlja njihovu udaljenost. U granici $r \to \infty$, operator treba težiti nekoj vrijednosti različitoj od nule za uređene sustave te nuli za neuređene sustave. Operator koji zadovoljava navedene uvjete dan je izrazom 2.8.

$$G(r) = \langle \sigma_i \sigma_j \rangle - \langle \sigma_i \rangle \langle \sigma_j \rangle. \tag{2.8}$$

Asimptotsko ponašanje korelacijske funkcije na visokim temperaturama opisuje se eksponencijalno padajućom funkcijom:

$$G(r) \sim \frac{1}{r^{\vartheta}} e^{-r/\xi}, \tag{2.9}$$

gdje je r udaljenost među spinovima, a ϑ eksponent čija vrijednost ovisi o tome je li sistem u uređenoj ili neuređenoj fazi. Sličan eksponencijalan pad primijećen je i na temperaturama ispod T_C. Algebarsko ponašanje u kritičnoj točki je:

$$G(r) \sim \frac{1}{r^{d-2+\eta}}, \tag{2.10}$$

gdje je η kritični eksponent, a d oznaka dimenzionalnosti sustava. Egzaktno rješenje

za 2D Isingov model s kratkodometnim interakcijama je $\eta = 0.25$, a ϑ između 0.5 i 2. Snižavanjem temperature smanjuju se termalne fluktuacije, a korelacijska dužina kreće divergirati u blizini faznog prijelaza [5]. Kako bi korelacijska dužina imala kontinuirani prijelaz iz konačne vrijednosti u beskonačnu vrijednost, ispod kritične točke mora vrijediti:

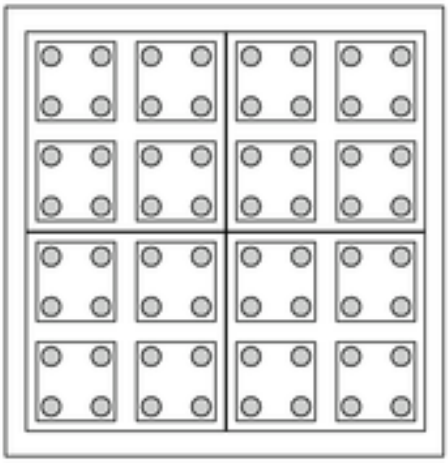
$$\xi \propto |T - T_C|^{-\nu}, \tag{2.11}$$

gdje je ν još jedan kritični eksponent.

Eksponencijalno ponašanje korelacijske funkcije odgovorno je za skaliranje koje se događa pri faznim prijelazima. Eksponenti su neovisni u temperaturi, štoviše, čak su jednaki za veliki spektar sustava.

2.2 Renormalizacijska grupa

U fizici, renormalizacijska grupa aparat je koji nam omogućava promatranje istog fizikalnog sistema kroz više različitih skala. Najjednostavniji primjer renormalizacijske grupe jesu blok spin grupe primijenjene na dvodimenzionalnom sustavu spinova [6].



Slika 2.2: Shematski prikaz blok spin renormalizacijske grupe primijenjene na 2D sustav [7].

Razmatra se sustav prikazan na slici 2.2. Neka spinovi interagiraju samo s najbližim susjedom i neka se sustav nalazi na temperaturi T. Ako je snaga interakcije kvantificirana konstantom vezanja J, tada se hamiltonijan sustava računa kao $H(T, J)$. Spinovi se grupiraju u blokove veličine 2×2. Sistem se opisuje preko blok varijabli, odnosno varijabli koje opisuju prosječno ponašanje bloka. Može se pretpostaviti da je fizika "novog" sustava opisana istim hamiltonijanom, ali s novim vrijednostima J' i T'. U tom slučaju,

renormalizirani sustav ima samo četvrtinu početnih spinova, ali zašto ne ponavljati proces sve dok se ne dođe do jednog gradivnog elementa sustava. Iz iteracije u iteraciju, skala promatranja sustava se povećava. Na početku, svaki spin se promatra zasebno, dok se nakon dvije iteracije promatra 16 spinova kao jednu cjelinu. Dodatno, svakom iteracijom redefiniraju se konstanta vezanja i temperatura koje nakon dovoljno puno iteracija konvergiraju u fiksnu točku.

Kod magnetskih sustava, J opisuje jakost poravnavanja susjednih spinova. Konfiguracija sistema rezultat je kompromisa između uređenja J i nereda uzrokovanih termalnim fluktuacijama. Razlikuju se 3 vrste fiksnih točaka [8]:

- $T = 0$ i $J \to \infty$. Temperatura je nebitna na većim skalama te dominira uređenje od vezanja J, odnosno sustav je u feromagnetskoj fazi.
- $T \to \infty$ i $J = 0$. Vezanje spinova nebitno je na većoj skali te dominira temperaturni efekt neuređenosti, odnosno sustav je u antiferomagnetskoj fazi.
- $T = T_C$ i $J = J_C$. U ovoj točki promjena skale ne mijenja fiziku sustava. Takvo stanje odgovara Curiejevom faznom prijelazu i naziva se kritična točka.

Formalno, postupak renormalizacije prikazan je kao:

$$H'(T', J') = \mathcal{R}H(T, J), \tag{2.12}$$

gdje je $\mathcal{R}$ renormalizacijski operator koji smanjuje broj stupnjeva slobode s N na N'. Tada se faktor skaliranja b definira kao:

$$b = \frac{N}{N'}. \tag{2.13}$$

Nužan uvjet koji transformacija mora zadovoljiti je nepromijenjenost particijske funkcije sustava (2.14).

$$Z(H') = Z(H) \tag{2.14}$$

Iz tog razloga, ukupna energija sustava se ne mijenja, ali se mijenja energija sustava po gradivnom elementu:

$$f(H') = bf(H). \tag{2.15}$$

Sve udaljenosti sustava smanjuju se za faktor b pa tako i korelacijska dužina koja se skalira kao:

$$\xi' = \frac{1}{b}\xi. \tag{2.16}$$

2.3 Isingov model

Isingov model, nazvan prema fizičaru Ernstu Isingu, matematički je model feromagnetizma. Diskretne vrijednosti (± 1) predstavljaju dva stanja magnetskog dipola: spin gore i spin dolje. Spinovi su složeni u rešetku koja može biti kvadratna, trokutasta, heksagonalna, itd. Poravnati spinovi imaju nižu energiju od onih međusobno suprotne orijentacije, što znači da svi spinovi imaju tendenciju biti orijentirani u istu stranu. Zbog termalnih fluktuacija, to nije uvijek moguće te dolazi do raznih strukturalnih fenomena (fazni prijelaz).

Najjednostavniji model uzima u obzir samo interakcije između susjednih spinova, no mogu se promatrati sustavi gdje je interakcija dugodosežna, ali opada s udaljenošću. Za najjednostavniji statistički sistem koji podliježe faznom prijelazu može se uzeti 2D Isingov model na kvadratnoj rešetci gdje spin interagira samo sa susjednim spinovima [9].

Razmatra se rešetka s N čvorova i periodičnim rubnim uvjetima. Sustav se nalazi na temperaturi T, a snaga interakcije među spinovima iznosi J. Omjer kritične temperature pomnožene s Boltzmanovom konstantom i snage interakcije takvog sustava iznosi $T_C k/J \approx 2.269$. U sustavu prirodnih jedinica dobijemo $T_C \approx 2.269$ K.

Aproksimacijom srednjeg polja može poslužiti kako bi se izračunala približna vrijednost nekih veličina sustava [10]. Hamiltonijan sustava je:

$$H = -h \sum_i \sigma_i - J \sum_{<i,j>} \sigma_i \sigma_j, \tag{2.17}$$

gdje je σ individualna vrijednost spina, a h vanjsko magnetsko polje. Energija jedne čestice dana je izrazom:

$$\epsilon(\sigma_j) = -h\sigma_j - J\sigma_j \sum_k \sigma_k, \tag{2.18}$$

gdje suma ide po susjedima spina j. Aproksimacija srednjeg polja ulazi u igru tako da se vrijednosti σ_k aproksimiraju njihovom prosječnom vrijednošću:

$$\epsilon_{mf}(\sigma_j) = -h\sigma_j - J\sigma_j \sum_k \langle \sigma_k \rangle = -h_{mf}\sigma_j \tag{2.19}$$

gdje je

$$h_{mf} = h + 4Jm. \tag{2.20}$$

m označava prosječnu magnetizaciju po spinu, a može se zapisati kao:

$$m = \frac{1}{N}\sum_i \langle \sigma_i \rangle. \tag{2.21}$$

Iz Boltzmannove distribucije proizlazi vjerojatnost da se spin j nađe u stanju σ (2.22).

$$p(\sigma_j) = \frac{e^{-\beta \epsilon_{mf}(\sigma_j)}}{\sum_{\sigma_i=\pm 1} e^{-\beta \epsilon_{mf}(\sigma_i)}} = \frac{e^{-\beta h_{mf}\sigma_j}}{e^{-\beta h_{mf}} + e^{\beta h_{mf}}} \tag{2.22}$$

Prosječna magnetizacija (uz uvjet da je vanjsko polje h jednako nuli) tada se može izračunati kao:

$$m = \sum_{\sigma_i=\pm 1} p(\sigma_j)\sigma_j = \tanh(4\beta J m) \tag{2.23}$$

3 Neuronske mreže

Neuronske mreže nastale su još 1960-ih, a njihovo ime, kao i njihova arhitektura, inspirirani su građom mozga. Mreže se sastoje od neurona koji su međusobno povezani pa je moguće da signal putuje s jednog kraja mreže na drugi. Nad signalom se vrše određene operacije, a izgled tih operacije uvelike je određen iznosima težina među neuronima. Težine su realni brojevi koji pojačavaju ili smanjuju signal na njihovom rubu, a te vrijednosti mreža uči za vrijeme treniranja.

Učenje neuronske mreže svodi se na to da mreža "probavi" puno primjera za koje se zna očekivani rezultat te prema tome prilagođava težine. Nakon dovoljnog broja iteracija, mreža može (do na neku točnost) predvidjeti izlaz za dani ulaz. Prednost ovakvih sistema je što se mogu primijeniti na veliki spektar problema bez prethodnog domenskog znanja. Sve što je potrebno su označeni podaci, a u nekim slučajevima nema potrebe niti za njima.

Primjer zadatka strojnog učenja jest napraviti algoritam koji prevodi tekst s engleskog na hrvatski jezik. Primijeni li se na dani tekst samo englesko-hrvatski rječnik, nastaje dosta grub prijevod nije ugodan za čitati, a često je i potpuno neispravan. Sljedeći korak kojim se nastoji poboljšati sustav dodavanje je posebnih pravila za prijevod. Na primjer, jedno pravilo može se odnositi na sprječavanje sustava da rečenicu *I listen to rock* ne prevede kao *Ja slušam kamen*. Kako postoji more ovakvih primjera, sustav bi vrlo brzo postao zatrpan pravilima koja je teško nadgledavati, održavati i skalirati.

S druge strane, pokuša li se ovaj problem riješiti neuronskim mrežama, sve što je potrebno jest skup tekstova koji imaju hrvatsku i englesku verziju. Prvo se izabere odgovarajuća arhitektura mreže, a potom se mreža uči neko vrijeme. Jednom kada mreža konvergira, sustav postaje konkurentan profesionalnim prevoditeljima. *Google translate* koristi ovu metodologiju kako bismo imali što kvalitetnije prijevode [11]. Ovo je samo jedna od mnnogih primjena neuronskih mreža u stvarnom svijetu. Štoviše, pokazale su se kao moćan alat u raznim područjima poput računalnog vida, obrade prirodnog jezika, otkrivanja novih lijekova, medicinskih dijagnoza, pa čak i slikanja. Neuronske mreže često se nazivaju univerzalnim funkcijama aproksimacije.

3.1 Od neurona do mreže

Osnovni gradivni blok svake mreže je neuron. Njegova glavna zadaća je dobiveni d-dimenzionalni ulazni vektor $\mathbf{x} = (x_1, x_2, ..., x_d)$ pretvoriti u skalarni izlaz $a(\mathbf{x})$. Dok je jedan sloj sastavljen od mnogo naslaganih neurona, mreža je sačinjena od više naslaganih slojeva pri čemu izlaz jednog sloja čini ulaz u drugi sloj (slika 3.1).

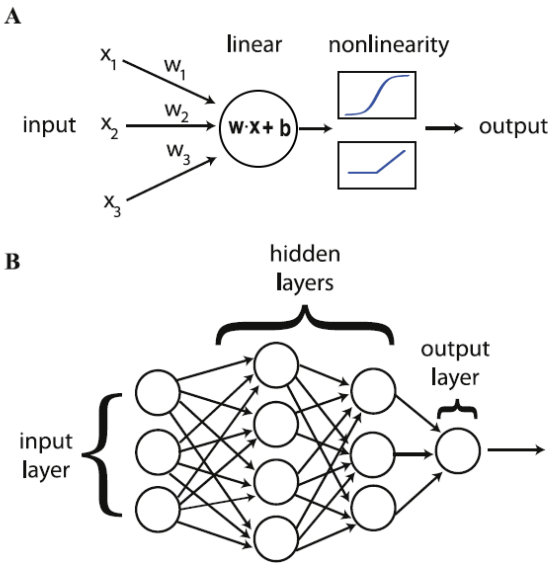
Točan izgled $a(\mathbf{x})$ ovisi o primijenjenoj aktivacijskoj funkciji. Jedini bitan uvjet je da funkcija mora biti nelinearna jer se u protivnom cijela mreža može prikazati linearnom regresijom čime se gubi na kompleksnosti informacije koja se nastoji naučiti. Svaki neuron ima pripadajuće težine $\mathbf{w} = (w_1, w_2, ..., w_d)$ popraćene slobodnim članom b.

Prolaz signala kroz neuron dan je izrazima 3.1 i 3.2.

$$z = \mathbf{w} \cdot \mathbf{x} + b = \mathbf{x}^T \cdot \mathbf{w} \tag{3.1}$$

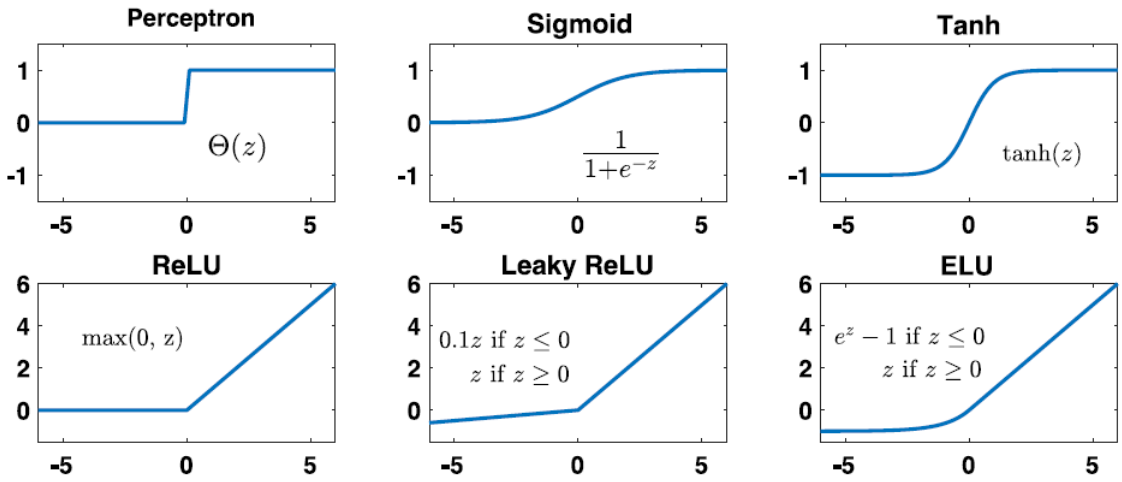
$$a(\mathbf{x}) = \sigma(z) \tag{3.2}$$

U izrazu 3.1, b je stavljen pod $\mathbf{w}$ kako bi se dobio elegantniji zapis, a $\mathbf{x}$ je redefiniran kao $\mathbf{x} = (1, \mathbf{x})$.



Slika 3.1: (A) Shematski prikaz neurona s težinama i aktivacijskom funkcijom. (B) Prikaz kako su neuroni međusobno povezani u mreži [12].

Kod jednadžbe 3.2, $\sigma(x)$ predstavlja bilo koju aktivacijsku funkciju koja unosi nelinearnost. Na slici 3.2 prikazane su neke od najčešće korištenih funkcija.



Slika 3.2: Najčešće korištene aktivacijske funkcije. U praksi prednost obično imaju derivabilne funkcije i to one čija je derivacija jednostavnija za izračunati, pa se tako donji red funkcija puno češće koristi [12].

Povijesno česti izbor aktivacijske funkcije bila je step funkcija. Razlog njezinog odabira je taj što rekreira paljenje i gašenje neurona baš kao što se u to mozgu događa. Izumom algoritma propagacije pogreške unazad [13], postalo je jasno da aktivacijske funkcije moraju biti derivabilne iz razloga što je polazna premisa deriviranje izlaza po težinama $\mathbf{w}$. Prirodni izbori zamjene step funkcije bili su *sigmoida* i *tanh* zbog derivabilnosti i sličnog izgleda same funkcije. Kako su modeli postajali sve dublji i kompleksniji, počeo se javljati problem iščezavajućeg gradijenta zbog množenja puno članova malog gradijenta. Kod *sigmoide* i *tanh*, mali gradijent javlja se upravo na rubovima funkcije. *ReLu, Leaky ReLU* te *ELU* funkcije su kojima se gradijent ne gubi čak ni prolaskom kroz puno slojeva mreže.

Cilj slaganja neurona u slojeve jedne iza drugih je dobiti hijerarhijsko ponašanje neurona. U slučaju obrade slika, neuroni u ranijim slojevima uče osnovne oblike poput horizontalnih i vertikalnih crta, dijelovi kruga i slično. Kombiniranjem znanja od prethodnih neurona, neuroni u dubljim slojevima sposobni su naučiti kompleksnije oblike kao što su lica, oči ili nos kada je riječ o brojanju ljudi na slici [14]. Odabir optimalne arhitekture za neki problem nije niti malo jednostavan zadatak i zahtijeva kompleksnu pretragu prostora arhitektura. Danas se zna koje su najpogodnije arhitekture za određene vrste problema, ali se često važe odluka između brzine učenja i točnosti modela.

3.2 Učenje mreže

Jednom kada je uvedena osnovna arhitektura neuronske mreže, bitno je razumjeti kako je efikasno učiti. Nadzirano učenje uglavnom se koristi za rješavanje dvije vrste problema, a to su klasifikacijski i regresijski problem. Kod klasifikacijskog zadatka cilj je svaki primjer svrstati u neku od ponuđenih klasa. Na primjer, sliku na kojoj je mačka želimo svrstati u kategoriju mačke. U regresijskom zadatku, za svaki primjer predviđa se kontinuirani broj (ili više njih). Jedan takav primjer jest predviđanje potražnje nekog proizvoda u danom razdoblju. Za oba pristupa potrebno je imati skup podataka s ulaznim primjerima i odgovarajućim izlazima.

Mrežu se optimizira minimiziranjem funkcije gubitka koja se mijenja ovisno o tome primjenjuje li se klasifikacija ili regresija. Za regresijske probleme, kao procjenu greške najčešće se koristi $L1$ ili $L2$ norma [12]. Greške su dane izrazima:

$$E(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i(\mathbf{x}_i, \mathbf{w})| \tag{3.3}$$

$$E(\mathbf{w}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i(\mathbf{x}_i, \mathbf{w}))^2}, \tag{3.4}$$

gdje je n broj primjera u skupu, y_i stvarna oznaka primjera $\mathbf{x}_i$, a $\hat{y}_i(\mathbf{x}_i, \mathbf{w})$ predikcija mreže s težinama $\mathbf{w}$ za primjer $\mathbf{x}_i$.

Kod klasifikacijskih problema, skup podataka je oblika $(\mathbf{x}_i, y_i)$, gdje y_i pripada jednoj od M klasa, odnosno: $y_i \in \{0, 1, ..., M-1, M\}$. Za svaki primjer i definira se *one-hot* vektor oblika:

$$y_{im} = \begin{cases} 1, & \text{ako } y_i = m \\ 0, & \text{inače.} \end{cases} \tag{3.5}$$

Funkcija pogreške za klasifikacijske probleme je unakrsna entropija definirana kao:

$$E(\mathbf{w}) = -\sum_{i=1}^{n}\sum_{m=0}^{M-1} y_{im} \log \hat{y}_{im}(\mathbf{w}, \mathbf{x}_i) + (1 - y_{im}) \log (1 - \hat{y}_{im}(\mathbf{w}, \mathbf{x}_i)). \tag{3.6}$$

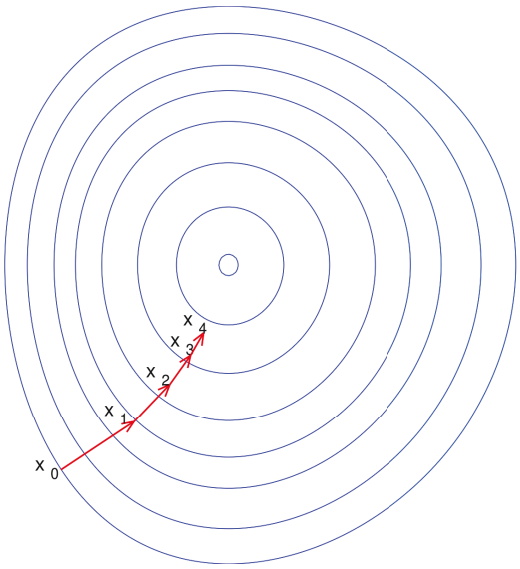
U funkcije pogreške često se dodaje i regularizacijski član koji ograničava težine da ne postanu prevelike, iz razloga što velike težine uzrokuju prenaučenost mreže. Veliki iskorak u regularizacijskim metodama napravljen je uvođenjem *Dropout* [15] sloja. *Dropout* sloj prilikom učenja nasumično postavlja težine na nulu čime se model tjera da za slične

primjere nađe više različitih putanja do sličnog izlaza. Veći broj putanja odgovara boljoj generalizaciji modela.

Nakon definicije arhitekture mreže i funkciju pogreške, zadnji korak je pronalazak optimalnih parametara (težina) modela. Cilj je minimizirati funkciju gubitka. Kako je nemoguće egzaktno naći optimalne parametre koji za dane podatke postižu najmanju grešku, koristi se gradijentni spust. Gradijentni spust [16] je optimizacijski algoritam koji se 'spušta' po funkciji u smjeru najvećeg negativnog gradijenta. Metoda ne garantira postizanje globalnog optimuma, no u praksi daje dovoljno dobre rezultate. Iterativno korigiranje parametra dano je jednadžbom 3.7, a shematski prikaz spusta prikazan je na slici 3.3.

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \xi \nabla E(\mathbf{w}_n) \tag{3.7}$$

ξ je stopa učenja koja definira koliko veliki korak se primjenjuje u svakoj iteraciji. Prevelika stopa učenja može prouzročiti divergenciju, dok optimizacija za malu stopu učenja traje predugo. Razumne vrijednosti ξ su od 10^{-6} za duboke mreže do 10^{-1} za plitke mreže. Postoje i naprednije vrste optimizatora poput SGD, *RMSprop*, *Adam* [17], *Adadelta* [18]. Dok se svi zasnivaju na ideji gradijentnog spusta, dodatno imaju određene korekcije poput "momentuma", prilagodljive stope učenja, i slično.



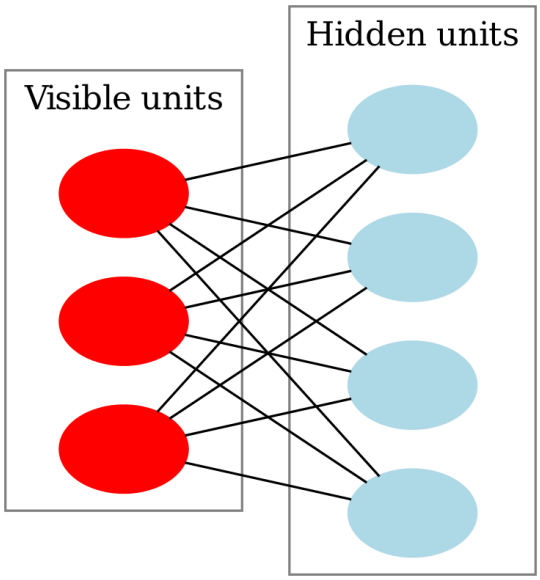
Slika 3.3: Ilustrativni prikaz gradijentog spusta, gušće linije simboliziraju veći nagib funkcije [19].

4 Generativni modeli

4.1 Ograničeni Boltzmannov stroj

Prethodno su opisani diskriminativni modeli, napravljeni kako bi razlikovali kategorije ili vrijednosti kojima pripada dana točka. Iako takvi modeli čine većinu nadziranog učenja, nemaju sposobnost generirati nove podatke, poput nove slike psa ili mačke. Kako bi se riješio taj problem, jedan je od načina okrenuti se generativnim modelima. To su modeli koji imaju sposobnost naučiti distribuciju na temelju koje su generirani podaci te je potom iskoristiti za generiranje novih podataka. Jedni od prvih generativnih modela bili su energetski modeli koji vuku dosta paralela sa statističkom fizikom.

Boltzmannovi strojevi energetski su modeli koji koriste vidljive i skrivene varijable v i h. Teorijski okvir može se izvesti nad najjednostavnijim Boltzmannovim strojem s jednim skrivenim slojem. Na temelju njega nadalje se lako primjenjuje generalizacija na dublje mreže s više skrivenih slojeva. Shematski prikaz takvog modela prikazan je na slici 4.1. Model mapira ulazni vektor $\mathbf{v}$ u skriveni vektor $\mathbf{h}$ preko matrice prijelaza $\mathbf{W}$. Također, model uči vektore $\mathbf{a}$, odnosno $\mathbf{b}$ koji su slobodni parametri za vidljivi, odnosno skriveni vektor. Ulazni su vektori, kao i skriveni vektori, binarizirani. Pritom se skriveni vektor postavlja na 0 ili 1 s određenom vjerojatnošću (izlaz iz sigmoide).



Slika 4.1: Shematski prikaz Boltzmannovog stroja [20].

Energija Boltzmannovog stroja dana je izrazom:

$$E(\mathbf{v}, \mathbf{h}) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_{ij} v_i W_{ij} h_j, \tag{4.1}$$

gdje prva dva člana podsjećaju na energiju interakcije između spinova i vanjskog polja, dok zadnji član modelira međusobnu interakciju vidljivih i skrivenih varijabli. Važno je naglasiti da ne postoje interakcije između samo skrivenih i samo vidljivih varijabli. To je analogno kvantnoj elektrodinamici gdje slobodni fermioni i fotoni interagiraju jedni s drugima, ali ne međusobno.

Model uči vektore **a**, **b** i matricu prijelaza **W**. Jednom kada se definira energija, može se izraziti vjerojatnost da model izgenerira određeni primjer. Vjerojatnosti uvelike pomažu prilikom učenja jer tjeraju model da s većom vjerojatnošću generira češće primjere. Time se osigurava cilj da model uči stvarnu distribuciju podataka.

Vjerojatnost da se mreža nađe u stanju **v**,**h** je:

$$p(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v},\mathbf{h})}}{Z}, \tag{4.2}$$

gdje je Z particijska funkcija koja se dobije zbrajanjem po svim kombinacijama **v** i **h**. Kako bi se dobila vjerojatnost da se generira neki vektor **v**, jednadžba 4.2 zbraja se po svim skrivenim stanjima [21]:

$$p(\mathbf{v}) = \frac{1}{Z} \sum_h e^{-E(\mathbf{v},\mathbf{h})}, \tag{4.3}$$

Također, vjerojatnost da se generira neki skriveni vektor **h** dana je kao:

$$p(\mathbf{h}) = \frac{1}{Z} \sum_v e^{-E(\mathbf{v},\mathbf{h})}. \tag{4.4}$$

Moguće je izračunati i uvjetnu vjerojatnost da se dobije **v** za dani **h** i obrnuto:

$$P(\mathbf{v}|\mathbf{h}) = \prod_{i=1}^{m} P(\mathbf{v}_i|\mathbf{h}), \quad \text{i} \quad P(\mathbf{h}|\mathbf{v}) = \prod_{j=1}^{n} P(\mathbf{h}_j|\mathbf{v}), \tag{4.5}$$

gdje je m broj vidljivih čvorova, a n broj skrivenih čvorova. Uvjetna vjerojatnost svakog

čvora posebno jednaka je:

$$P(v_i|h) = \sigma(a_i + \sum_{j=1}^{n} w_{i,j}h_j) \text{ i } P(h_j|v) = \sigma(b_j + \sum_{i=1}^{m} w_{i,j}v_i). \tag{4.6}$$

Boltzmannovi strojevi uče maksimiziranjem očekivanih logaritam vjerojatnosti generiranja podataka iz skupa za učenje V [12],

$$L = \langle \log P(\mathbf{v}) \rangle_V. \tag{4.7}$$

Greška se optimizira gradijentnim spustom, a gradijenti po danim varijablama tada iznose:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}} &= \mathbf{v}\mathbf{h}^\intercal - \mathbf{v}'\mathbf{h}'^\intercal \\ \frac{\partial L}{\partial \mathbf{a}} &= \mathbf{v} - \mathbf{v}' \\ \frac{\partial L}{\partial \mathbf{b}} &= \mathbf{h} - \mathbf{h}', \end{aligned} \tag{4.8}$$

gdje su $\mathbf{v}'$ i $\mathbf{h}'$ vidljivi i pripadajući skriveni vektori generiranih podataka.

Budući da je generiranje podataka za sve kombinacije skrivenog vektora računalno zahtjevna operacija, javlja se problem kako efektivno uzrokovati nove podatke. Tada se koristi aproksimativna metoda zvana Gibbsovo uzorkovanje. Za dani ulazni vektor, izračunava se skriveni vektor i potom koristi kako bi se nanovo dobio vidljivi vektor. Ovaj proces se ponavlja k puta, a u praksi se često odabire vrijednost $k = 1$.

4.2 Varijacijski autoenkoder

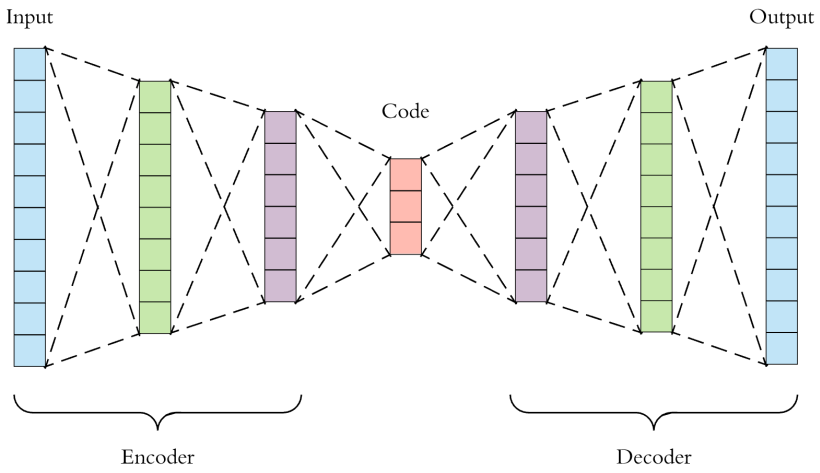
Energetski modeli daju dobre rezultate kada se na ulaz postave jednostavni podaci, ali kod generiranja teksta ili realnih ljudskih lica podbace. Dodatno, uzme li se u obzir činjenica da njihov proces učenja nije najjednostavniji, postaje jasno da se u praksi naginje drugim generativnim modelima.

Najbolji kompromis između generativne moći i jednostavnosti učenja daje varijacijski autoenkoder (VAE). VAE je nadogradnja običnog autoenkodera kako bi ga se prenamijenilo iz učenja latentnih vektora podataka u generiranje podataka preko istih.

Autoenkoderi su modeli koji se koriste za smanjenje dimenzionalnosti podataka, a jedna njihova definicija temelji se na nelinearnoj analizi svojstvenih komponenti (engl.

Principal component analysis (PCA)) [22]. PCA je ortogonalna linearna transformacija koja transformira podatke u novi koordinatni sustav manje dimenzionalnosti od početnog, uz što veće očuvanje informacije. To znači da vraća n vektora (ovisno o dimenzionalnosti u koju se mapiraju podaci) koji najbolje opisuju podatke u tom prostoru. Može ih se smatrati zapravo svojstvenim vektorima sustava. Autoenkoderi rade istu stvar samo što mapiranje u niži prostor nije linearno, čime se postiže veća ekspresivna moć.

Model se sastoji od enkodirajućeg i dekodirajućeg dijela koji su povezani preko srednjeg sloja. Arhitektura modela je prikazana na slici 4.2. Oba dijela čini neuronska mreža koja se sastoji od nekoliko skrivenih slojeva s proizvoljnom aktivacijskom funkcijom. Učenje modela odvija se tako da se minimizira srednja kvadratna pogreška između rekonstruiranih podataka i ulaznih podataka. Iako je autoenkoder dobar u izvlačenju latentnih vektora, problem se javlja u tome što je distribucija latentnih vektora nepoznata. To za posljedicu ima nemogućnost generiranja podataka, odnosno generiranja latentnih vektora iz kojih će model rekonstruirati podatke.



Slika 4.2: Shematski prikaz autoenkodera [23].

Varijacijski autoenkoder zahtijeva da vektor srednjeg sloja prati normalnu distribuciju. To ograničenje je nadodano funkciji pogreške u obliku Kullback–Leiblerove (KL) divergencije [24]. Kako je srednji sloj probabilistički, onda je i sam dekoder također probabilistički. Iz tog razloga, potrebna je prilagodba pogreške rekonstrukcije kako bi se maksimizirala vjerojatnost dobre rekonstrukcije za dani skriveni vektor z. Ukupna funkcija pogreške VAE-a tada izgleda kao:

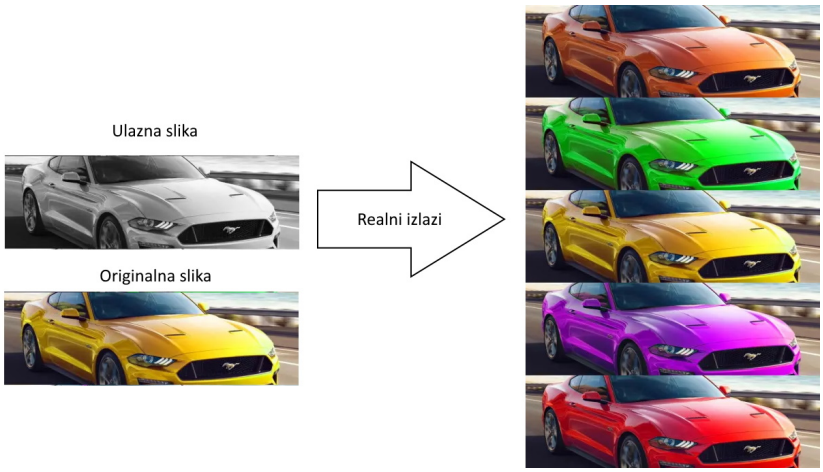
$$L = \frac{1}{2}[1 + \log(\sigma_j^2) - \sigma_j^2 - \mu_j^2] + \log p(x_i|z), \tag{4.9}$$

gdje su σ_j i μ_j varijanca i prosjek j-te komponente vektora z, a $p(x_i|z)$ izlaz iz modela za i-ti primjer. Prvi dio izraza proizlazi iz KL divergencije, a drugi dio iz rekonstrukcijske pogreške.

4.3 Generativne suparničke mreže

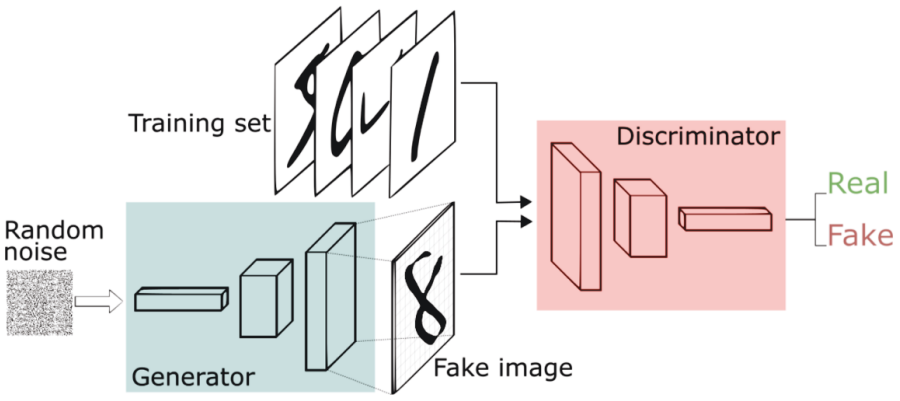
Generativne suparničke mreže [25] (engl. generative adversarial networks, GAN) smatraju se *state-of-the-art* modelima za generativne probleme. GAN-ovi nastoje generirati što realnije primjere.

Na vrlo jednostavnom primjeru može se pokazati zašto su GAN-ovi superiorniji od VAE-ova. Slučaj izgradnje modela koji treba crno bijele slike pretvoriti u slike u boji vidljiv je na slici 4.3. Istu sliku može se obojiti na više načina od kojih je svaki realan, pa se ne može jednoznačno odrediti kako bi to uistinu zapravo trebalo izgledati. Primjenom VAE na ovakvom problemu, zahtjeva se da auto bude žute boje, te se samim time lošije generalizira i teže uči. Dođe li slika poput slike 4.3, na kojoj je auto neke druge boje, VAE se tu muči jer dva slična ulaza daju dosta različiti izlaz. Povuče li se analogija s fizikom, model nije otporan na male perturbacije što je definicija prenaučenosti. GAN-ovima, s druge strane, je bitno da obojana slika izgleda realno neovisno o boji originalne slike. Takav pristup rješava problem kod kojih autoenkoderi pucaju.



Slika 4.3: Primjer za koji jedan ulaz ima više realnih izlaza.

Implementacijski, GAN se sastoji od dvije mreže: generatora i diskriminatora kao što je prikazano na slici 4.4. Generator iz šuma ili referentne točke generira primjer, dok diskriminator za ulazni primjer mora odrediti je li on stvaran ili generiran. Generator je u praksi dekoder ili autoenkoder, dok je diskriminator obični binarni klasifikator.



Slika 4.4: Shematski prikaz arhitekture GAN-a [26].

Generator i diskriminator uče nasumično propagacijom gradijenta unatrag. Funkcije gubitka za diskriminator i generator dane su jednadžbama 4.10 i 4.11, gdje je $D(x)$ izlaz iz diskriminatora, $G(x)$ izlaz iz generatora, a z ulazni šum (ili primjer) u generator.

$$L_D = -\log D(x) - \log\left(1 - D(G(z))\right) \tag{4.10}$$

$$L_G = \log\left(1 - D(G(z))\right) \tag{4.11}$$

Cilj diskriminatora je što bolje naučiti razlikovati prave i generirane slike, dok generator nastoji prevariti diskriminator kako bi za generirane slike rekao da su prave. Kod učenja GAN-a, ne može se postići da obje funkcije gubitka padaju. Kako jedna funkcija pada, druga raste iz razloga što imaju isti član s različitim predznakom. Ako dođe do slučaja da jedna pogreška značajno pada, to znači da je diskriminator (ili generator) postao predobar što globalu daje loše rezultate. Iz tog razloga, cilj je obje funkcije pogreške konstantno držati na vrijednostima sličnim početnima.

4.4 Tok renormalizacijske grupe i generativni modeli

Dosta radova pokazalo je da se neuronske mreže ponašaju kao jedna vrsta renormalizacijske grupe (RG) [27] [28] [29]. Analogija se može pokazati na primjeru RBM-a te kako njegova uzastopna primjena ima slično ponašanje kao i uzastopna primjena varijacijske renormalizacijske grupe.

Sistem je opisan vektorom v i hamiltonijanom $H(v)$. Cilj je mapirati sistem u drugu skalu u kojoj će biti opisan vektorom h i novim hamiltonijanom $H^{RG}(h)$. Transformacija

se provodi preko operatora $T(v,h)$ koji ovisi o parametrima λ. Hamiltonijan nakon RG transformacije dan je izrazom:

$$e^{-H_\lambda^{RG}(h)} = \sum_v e^{T_\lambda(v,h)-H(v)}. \tag{4.12}$$

Za odrediti parametre λ, potrebno je varirati $T(v,h)$ po λ kako bi se minimizirao izraz:

$$\log\left(\sum_v e^{-H(v)}\right) - \log\left(\sum_h e^{-H_\lambda^{RG}(h)}\right). \tag{4.13}$$

Definira li se $T_\lambda(v,h) = -E(v,h) + H(v)$ te raspiše vjerojatnost da se sustav nalazi u stanju h, nastaje izraz:

$$\frac{e^{-H_\lambda^{RG}(h)}}{Z} = \frac{\sum_v e^{T_\lambda(v,h)-H(v)}}{Z} = \frac{\sum_v e^{-E(v,h)}}{Z} = p_{RBM}(h) = \frac{e^{-H^{RBM}(h)}}{Z}, \tag{4.14}$$

gdje se predzadnjoj jednakosti koristi jednadžba 4.4 te naposljetku definira hamiltonijan RBM-a.

Jednadžba 4.14 pokazuje da je hamiltonijan RG-a isto što i hamiltonijan RBM-a. Također, može se pokazati da minimizacija uvjeta 4.13 odgovara minimizaciji KL divergencije. Da je iterativna primjena modela isto što i tok RG-a, prikazano je isključivo na RBM-u. Dodatno, metoda se primjenjuje na VAE i GAN kako bi se istražilo daju li napredniji modeli bolje rezultate.

5 Implementacija problema na Isingovom modelu

5.1 Generiranje podataka

Cilj je generirati, pomoću Monte Carlo metode, 2D spinske sustave na raznim temperaturama. Podaci se generiraju za tri vrste rešetke: kvadratnu, trokutastu i heksagonalnu. Dimenzija rešetki fiksirana je na $L \times L = 20 \times 20$. Korištene su temperature od 0 do 6 K s povećanjem od 0.25 K (25 temperatura). Za svaku temperaturu generirano je po 1000 primjera u skupu za učenje i 1000 primjera u skupu za testiranje čime svaki skup ima ukupno 25000 primjera. Konstante J i k_B postavljene su na 1 iz razloga što to ne utječe na fizikalnu pozadinu.

Postupak simuliranja podataka za danu temperaturu T je sljedeći:

1. Stvori se nasumična matrica dimenzija $L \times L$ ispunjenu s 1 ili -1.

2. Nasumično se izabere jedan spin $\sigma_{x,y}$ (koordinate x, y) iz matrice te izračuna promjenu energije $dE_{x,y}$ u slučaju da se okrene izabrani spin. Bitno je napomenuti da struktura rešetke utječe samo na $dE_{x,y}$. Nametnut je rubni uvjet $\sigma_{1,i} = \sigma_{L,i}$ za $i \in \{1, 2, ..., L-1, L\}$ i $\sigma_{j,1} = \sigma_{j,L}$ za $j \in \{1, 2, ..., L-1, L\}$ kako bi se dobilo ponašanje beskonačne rešetke.

3. Izračuna se vjerojatnost okretanja spin x, y prema jednadžbi:

$$p_{x,y} = \begin{cases} 1 & dE_{x,y} \leq 0 \\ e^{-dE_{x,y}/T} & dE_{x,y} > 0 \end{cases} \tag{5.1}$$

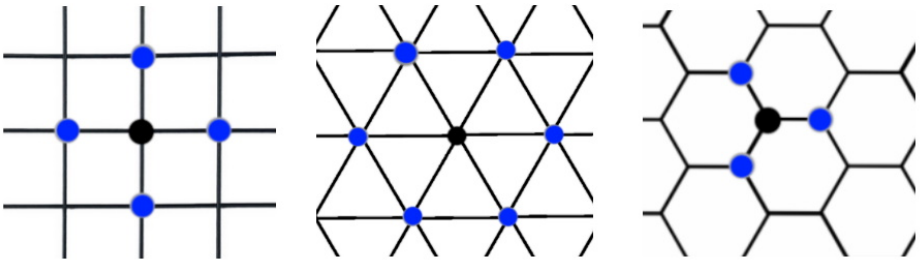
4. Točke 2 i 3 se ponavljaju $100L^2 = 40000$ puta.

Promatraju se samo interakcije s najbližim susjedima. No ovisno o rešetci, broj susjeda je drukčiji. Promjena energije dana je izrazom:

$$dE_{x,y} = 2\sigma_{x,y} \dot{\sum}_{i,j} \sigma_{i,j}, \tag{5.2}$$

gdje suma ide po susjedima točke x,y. Na slici 5.1 su shematski prikazane rešetke te susjedi za označenu česticu.

Cijeli postupak generiranja podataka implementiran je u `Python`-u [30] uz pomoć paketa `numpy` [31]. U što većoj mjeri se koristilo vektorsko pisanje koda kako bi što bolje



Slika 5.1: Shematski prikaz rešetki i susjeda (plavo) centralnog (crno) čvora za: a) kvadratnu rešetku, b) trokutastu rešetku i c) heksagonalnu rešetku.

iskoristili ubrzanje `numpy`-a u odnosu na *for* petlje. Postupak generiranja 25000 podataka za jednu rešetku traje 25 minuta na *Ryzen 2700X* procesoru s 32 GB RAM-a.

5.2 Modeli

Pristup zahtijeva dva modela: klasifikacijski i generativni. Klasifikacijski model koristi se kako bi se odredila distribucija vjerojatnosti temperature sustava, dok se generativni model koristi kako bi se generiralo iduće stanje sustava. Svi modeli su napisani u *pytorchu* [32] i učeni su na grafičkoj kartici *RTX 2070*.

Postupak se sastoji od pet koraka:

1. Nauči se klasifikacijski model za određivanje temperature - temperaturni model.
2. Nauči se generativni model (RBM, VAE ili GAN).
3. Izabere se sustav iz skupa za testiranje, na nekoj temperaturi te se provuče kroz generativni model da se dobije novi sustav. Novom sustavu se "izmjeri" temperatura (najvjerojatnija točka distribucije).
4. Korak 3 se ponavlja 10000 puta, no sada se ne uzima sustav iz skupa za testiranje nego sustav koji je izgeneriran u prethodnom koraku.
5. Očita se temperatura nakon koraka 4 te se usporedi koliko odstupa od stvarne kritične temperature.

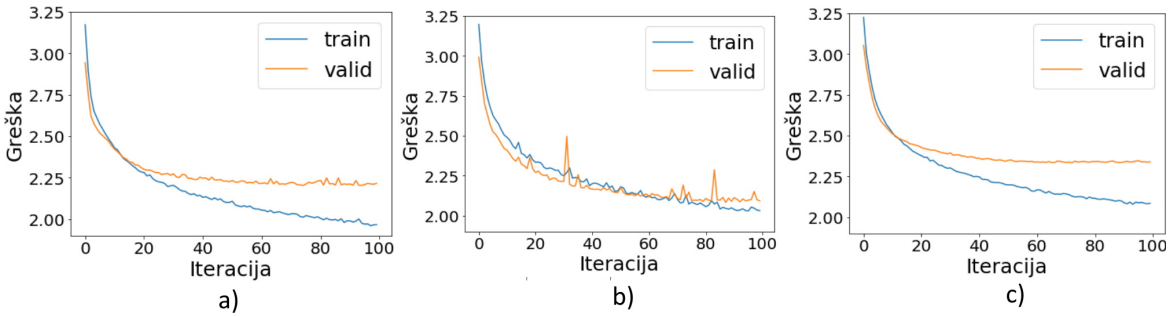
5.2.1 Temperaturni model

Iako su podaci u 2D, odnosno prikazani pomoću matrica, za sve modele su razmotani u vektor veličine L^2. Temperaturni model je potpuno povezana mreža koja se sastoji

od 7 slojeva ukupno. U *pytorch*-u potpuno povezani sloj (objašnjen u poglavlju 3.2) bez aktivacijske funkcije se zove *Linear* sloj, pa je u nastavku korištena takva terminologija. Također, koristi se *BatchNorm* [33] sloj koji pomaže u generaliziranju i *Dropout* sloj koji je definiran s postotkom težina koje će ugasiti za vrijeme učenja. Cijela arhitekura mreže je:

$$\begin{aligned} &Linear(L^2, 64) \rightarrow ELU \rightarrow BatchNorm1d \rightarrow Dropout(0.25) \rightarrow \\ &Linear(64, 64) \rightarrow ELU \rightarrow BatchNorm1d \rightarrow Dropout(0.25) \rightarrow Linear(64, 25) \end{aligned} \tag{5.3}$$

Funkcija pogreške je unakrsna entropija. Bitno je primijetiti da na izlaznom sloju ne postoji *softmax* aktivacijska funkcija koja se obično koristi kod klasifikacije. Razlog tome je što korištena funkcija pogreške u sebi ima implementiran *softmax* zbog numeričke stabilnosti. Model se učio 100 iteracija uz pomoć optimizatora *Adam* sa stopom učenja 0.001. Slika 5.2 prikazuje funkciju pogreške kroz epohe za sve tri rešetke.



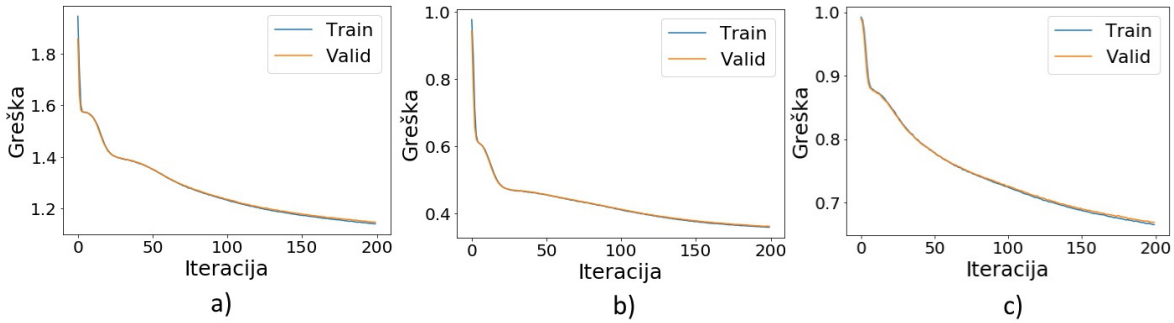
Slika 5.2: Funkcija pogreške temperaturnog modela na skupu za učenje i validaciju kroz iteracije za: a) kvadratnu rešetku, b) trokutastu rešetku i c) heksagonalnu rešetku.

5.2.2 Ograničen Boltzmannov stroj

Ograničen Boltzmanov stroj sastoji se od jednog skrivenog sloja veličine 64, dok je ulazni vektor veličine 400. Zbog prirode problema sve aktivacije su *tanh*. Početna inicijalizacija vektora težina a i b je takva da su svi elementi postavljeni na nulu, dok je matrica prijelaza W inicijalizirana normalnom distribucijom s prosjekom 0 i standardnom devijacijom 0.01. Model se učio 200 iteracija sa stopom učenja 0.0001.

Kako bi se pratilo učenje modela, prati se pogreška rekonstrukcije podataka. Iako je cilj modela naučiti distribuciju podatka, indirektno se treba smanjivati pogreška rekonstrukcije. Ako se ta pogreška ne smanjuje, to je signal da nešto ne radi kako treba. Na slici 5.3

je prikazana ovisnost pogreške rekonstrukcije o iteraciji za sve tri rešetke.



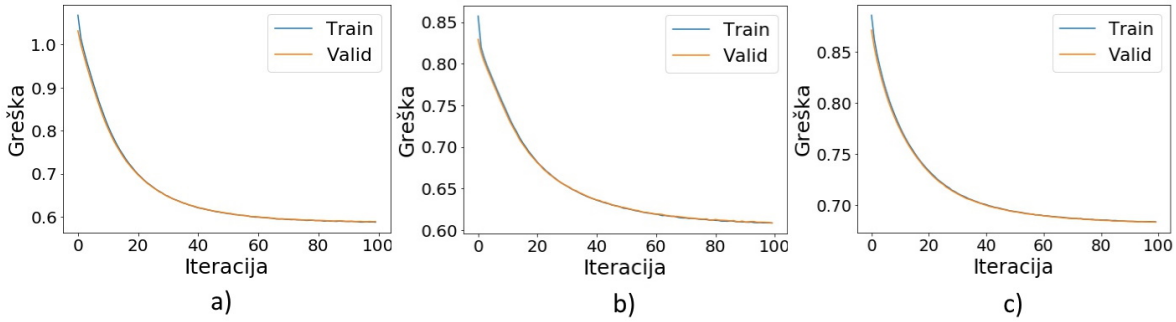
Slika 5.3: Pogreška rekonstrukcije RBM-a na skupu za učenje i validaciju kroz iteracije za: a) kvadratnu rešetku, b) trokutastu rešetku i c) heksagonalnu rešetku.

5.2.3 Varijacijski autoenkoder

VAE prilikom prolaska podatka put naprijed radi nasumično uzorkovanje po naučenoj distribuciji srednjeg sloja. Kako bi se osigurao prolazak gradijenta unatrag kroz takav sloj, mora se primijeniti mudra implementacija. Problem se rješava tako da, od dva ulazna sloja, jedan se proglasi prosjekom μ distribucije, a drugi standardnom devijacijom σ. Izlaz iz skrivenog sloja dan je izrazom $z = \mu + \epsilon \cdot \sigma$, gdje je ϵ biran nasumično iz standardne normalne distribucije. Ovakva implementacija omogućava prolaz gradijenta unatrag jer sadrži samo aditivne i multiplikativne elemente. Ulazni sloj je $Linear(L^2, 128)$ s ELU aktivacijom koji se grana na dva $Linear(128, 64)$ sloja: jedan za prosjek drugi za standardnu devijaciju. Dekoderski dio je dan arhitekturom: $Linear(64, 128) \rightarrow ELU \rightarrow Linear(128, L^2) \rightarrow tanh$. Model se učio 100 iteracija pomoću optimizatora $Adamax$ sa stopom učenja 0.0001. Na slici 5.4 prikazana je ovisnost funkcije pogreške o iteraciji za sve tri rešetke.

5.2.4 Generativna suparnička mreža

Kao što je već spomenuto, GAN-u je cilj generirati realne slike. Svi sustavi koji su u ovom radu promatrani izgledaju isto na temperaturi od 0 K, a to je stanje da su svi spinovi orijentirani u istom smjeru. Kako skup podataka sadrži sustave na temperaturi od 0 K, generator je brzo naučio da treba generirati samo primjere gdje su svi spinovi isti. Gledano s tehničke strane, to je točno rješenje, ali nema koristi od modela koji uvijek generira isti ishod. Kako bi se generator natjerao da generira raznolike temperature, funkciji



Slika 5.4: Funkcija pogreške VAE-a na skupu za učenju i validaciju kroz iteracije za: a) kvadratnu rešetku, b) trokutastu rešetku i c) heksagonalnu rešetku.

pogreške generatora (4.11) nadodan je član koji zahtijeva da prosječan spin ulaznog i generiranog sustava budu što sličniji. Novi član je kvadratno odstupanje prosječnog spina generiranog i ulaznog sustava.

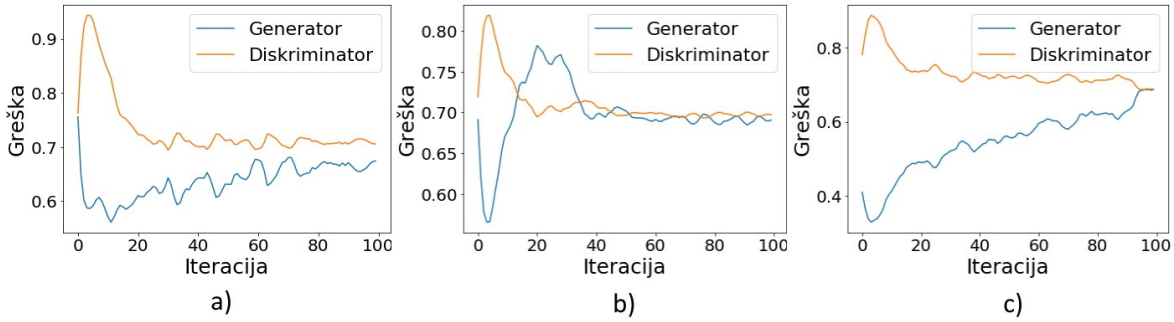
Arhitektura generatora dana je izrazom:

$$\begin{aligned} &Linear(L^2, 128) \rightarrow Sigmoida \rightarrow Linear(128, 64) \rightarrow Sigmoida \rightarrow \\ &Dropout(0.2) \rightarrow Linear(64, 128) \rightarrow Sigmoida \rightarrow Linear(128, L^2) \rightarrow Tanh. \end{aligned} \tag{5.4}$$

Diskriminator je definiran arhitekturom:

$$Linear(L^2, 64) \rightarrow Sigmoida \rightarrow Linear(64, 1) \rightarrow Sigmoida. \tag{5.5}$$

Generator i diskriminator uče se sa zasebnim optimizatorima gdje su oba optimizatora *Adam*. Stopa učenja generatora je 0.0001, a diskriminatora 0.00003. Broj iteracija je 100, no bitno je naglasiti da se diskriminator uči s vjerojatnošću od 20% u svakoj iteraciji. Učenje diskriminatora u svakoj iteraciji uzrokuje previše realno razlikovanje podataka čime se generatoru ne daje prostor za učenje. U tom slučaju, razlog zašto ne može učiti bolje generirati sustave jest što je potreban preveliki skok u jednoj iteraciji. Na slici 5.5 prikazana je ovisnost funkcije pogreške o iteraciji za sve tri rešetke. Može se primijetiti da se pogreške diskriminatora i generatora uvijek natječu što je i cilj kod ovakvih modela.



Slika 5.5: Funkcije pogreške generatora i diskriminatora na skupu za učenje kroz iteracije za: a) kvadratnu rešetku, b) trokutastu rešetku i c) heksagonalnu rešetku.

5.3 Tok renormalizacije grupe

Tok renormalizacijske grupe postiže se tako da se izabere početna temperatura T_{start}, te se iz skupa za validaciju nasumično izabere 50 primjera na toj temperaturi. Primjeri su ulaz u jedan od generativnih modela. Novonastalim točkama mjeri se temperatura uz pomoć temperaturnog modela. Temperaturni model vraća distribuciju po temperaturama, a odabire se ona točka s najvećom vjerojatnošću kao temperatura tog sustava. Nadalje, novonastali primjeri su ulazi u generativni model i cijeli postupak se ponavlja 10000 iteracija. Za temperaturu konvergencije uzima se prosječna temperatura zadnjih 1000 iteracija.

U tablici 5.1 su prikazane krajnje temperature razne početne temperature, kao i prosječna krajnja temperatura preko svi početnih temperatura T_{avg}^{final}. Vidljivo je da u većini slučajeva T_{avg}^{final} odgovara kritičnoj temperaturi sustava. RBM ima konzistentne rezultate

		T_{start} [K]								
Generator	Rešetka	0	1	2	3	4	5	6	T_{avg}^{final} [K]	$T_{true}^{critical}$ [K]
RBM	□	2.249	2.250	2.250	2.249	2.250	2.250	2.250	2.249	2.269
RBM	△	3.434	3.452	3.422	3.417	3.459	3.423	3.449	3.437	3.641
RBM	⬡	1.500	1.500	1.500	1.500	1.500	1.500	1.500	1.500	1.519
VAE	□	2.247	2.249	2.248	2.248	2.249	2.249	2.248	2.248	2.269
VAE	△	3.738	3.731	3.739	3.734	3.743	3.739	3.741	3.738	3.641
VAE	⬡	5.753	5.733	5.746	5.724	5.749	5.729	5.746	5.740	1.519
GAN	□	1.997	1.956	1.870	2.000	2.000	2.000	2.000	1.974	2.269
GAN	△	3.500	3.500	3.500	3.500	3.500	3.500	3.500	3.500	3.641
GAN	⬡	5.749	5.748	5.749	5.749	5.749	5.749	5.748	5.749	1.519

Tablica 5.1: Tablica rezultata toka renormalizacijske grupe. Vrijednost predstavlja temperaturu konvergencije, dane su sve kombinacije generatora i rešetke kao i 7 početnih temperatura.

preko svih rešetki. VAE ima manje odstupanje za trokutastu rešetku, no za heksagonalnu

rešetku ima odstupanje za više od faktora 3. GAN, iako idejno najnapredniji model, također daje loše rezultate za heksagonalnu rešetku. Dodatno, kada se uzme u obzir da učenje GAN-a zahtijeva dosta namještanja stopa učenja, vjerojatnosti učenja u iteraciji i ostalih parametara. Postaje jasno da je puno bolje koristiti RBM ili čak VAE za neke sustave. Svi modeli daju konzistentnu temperaturu konvergencije neovisno o početnoj temperaturi sustava.

5.4 Tok bez generatora

U znanosti nije strano da se do određenih znanstvenih otkrića poput X zraka, radioaktivnosti i inzulina dođe slučajno. Ovaj dio rada je nastao kao posljedica *bug*-a u kodu koji se na kraju pokazao korisnim. *Bug* je specifičan za sloj *BatchNorm* koji se drukčije ponaša za vrijeme učenja i evaluacije (ako mu se to naglasi).

Poznato je da prilikom učenja neuronske mreže podatke treba skalirati ili normalizirati kako bi mreža lakše učila. Korišteni podaci imaju vrijednosti 1 ili -1 pa nije bilo potrebe za tim. Međutim, ne postoji garancija da će podaci ostati normalizirani dok prolaze iz sloja u sloj mreže. Ako je dobro normalizirati ulazne podatke u mrežu, zašto se onda ne bi normalizirao i ulaz u svaki sloj.

BatchNorm radi upravo to tako što prati prosjek μ i varijancu σ podatka koji prođu kroz njega te ih skalira kao:

$$\hat{x}_i^k = \frac{x_i^k - \mu^k}{\sqrt{\sigma^{k^2} + \epsilon}}, \tag{5.6}$$

gdje i označava redni broj primjera, k govori koja je komponenta vektora, a ϵ je mala konstanta zbog numeričke stabilnosti. Kada se koristi model koji ima *BatchNorm* sloj, tada ga treba "zamrznuti" za vrijeme predikcije kako ne bi osvježavao prosjek i varijancu novim podacima. U slučaju kada se sloj ne zamrzne, a par puta obavi predikcija za isti primjer, ta predikcija će svaki put biti drukčija. Naravno, validacijom će se onda utvrditi da takav model ima izrazito nisku točnost. Postupak je sljedeći:

1. Nauči se klasifikacijski model za određivanje temperature - temperaturni model.
2. Izabere se 50 sustava određene temperature iz skupa za testiranje.
3. Temperaturni model se ostavi u načinu rada za učenje tako da se prosjek i varijanca *BatchNorm* slojeva ažurira (*napomena: u svim prethodnim analizama model je bio prebačen u način za evaluaciju*).

4. Napravi se mjerenje temperature za sustave 20000 puta. Izmjerena temperatura je ona s najvećom vjerojatnošću po distribuciji koju predvidi temperaturni model.

5. Očita se temperatura konvergencije tako da se uzme prosječnu temperaturu zadnjih 10000 iteracija.

Arhitektura i način učenja temperaturnog modela isti su kao i u potpoglavlju 5.3.1.

	T_{start} [K]								
Rešetka	0	1	2	3	4	5	6	T_{avg}^{final} [K]	$T_{true}^{critical}$ [K]
□	2.276	2.361	2.278	2.274	2.273	2.276	2.275	2.287	2.269
△	3.504	3.573	3.544	3.634	3.661	3.902	3.647	3.638	3.641
⬡	1.753	1.745	1.772	1.767	1.760	1.757	1.879	1.776	1.519

Tablica 5.2: Tablica rezultata konvergencije temperaturnog modela. Vrijednost predstavlja temperaturu konvergencije za sve tri rešetke kao i za 7 početnih temperatura.

Rezultati su prikazani tablicom 5.2. Ovaj pristup daje najbolji rezultat za trokutastu rešetku u odnosu na ostale pristupe. Najveće odstupanje od 17% je za heksagonalnu rešetku. Globalno rezultati su bolji nego kod VAE i GAN-a, no nešto lošiji nego kod RBM-a.

Tijekom generiranja podataka i učenja modela nikad se nije unijela informacija o kritičnoj temperaturi sustava. To znači da ovaj pristup ne iskorištava informacije koje ne bi smio koristiti. Hipoteza je da konstantna primjena normalizacije skrivene vektore svede na oblik vektora nasumičnog sustava. Prosjek nasumičnih sustava ima za svaku komponentu iznos nula (jer se međusobno skrate), a primjena normalizacije radi upravo to. Naravno treba nekoliko iteracija da μ i σ dođu do prave vrijednosti.

Ako se fazni prijelaz promatranih sustava karakterizira kao prijelaz iz uređenog u neuređeno (kaotično) stanje, tada vektor nasumičnog sustava se kategorizira u temperature $T > T_c$. Ostaje nejasno zašto konvergira u $T \approx T_c$, a ne u neke više temperature gdje je sustav također neuređen. Bilo bi zanimljivo istražiti točnost ove metode na sustavima čiji se fazni prijelaz ne karakterizira kao prijelaz u neuređeno stanje.

6 Zaključak

U ovo radu testirana je metoda za pronalaženje kritične temperature fizikalnih sustava pomoću strojnog učenja. Metoda je primijenjena na 2D spinskim sustavima. Generirani su podaci za spinske rešetke dimenzija 20×20 na temperaturama od 0 K do 6 K s porastom od 0.25 K. Za rešetke su izabrane kvadratna, trokutasta i heksagonalna rešetka. Za spinsku interakciju je izabrana interakcija između najbližih susjeda kojih je 3, 4 ili 6 (ovisno u rešetci).

Nakon generiranja podataka slijedilo je učenje modela za temperaturu. Model za temperaturu neuronska je mreža koja za ulazni sustav predviđa distribuciju vjerojatnosti temperature sustava.

Pokazano je da generativni modeli imaju dosta poveznica s renormalizacijskom grupom. Korištena su tri generativna modela: ograničen Boltzmannov stroj, varijacijski autoenkoder i generativne suparničke mreže. Nakon što svaki od njih zasebno nauči distribuciju podataka, slijedi iterativno generiranje podataka (tok renormalizacijske grupe). Za vrijeme iterativnog generiranja pratile su se temperature u koje konvergiraju novonastali sustavi.

Također, ispitano je kako početna temperatura sustava utječe na temperaturu konvergencije. Dobivene temperature su u većini slučajeva odgovarale kritičnim temperaturama sustava. Ograničen Boltzmannov stroj konzistentno je pružao dobre rezultate za sve tri rešetke. Varijacijski autoenkoder i generativne suparničke mreže dali su dobre rezultate na kvadratnoj i trokutastoj rešetci, dok su na heksagonalnoj rešetci imali prevelika odstupanja od kritične temperature. Svaka rešetka imala je svoje temperaturne i generativne modele, ali su svi modeli imali jednaku arhitekturu.

Dodatno, uvedena je nova metoda koja se zasniva na uzastopnoj normalizaciji vektora u skrivenim slojevima modela za temperaturu. Metoda ne zahtijeva generator jer koristi ponašanje normalizacijskih slojeva u temperaturnom modelu. Ovaj pristup daje bolje rezultate od VAE-a i GAN-a za sve tri rešetke i zahtijeva samo jedan model. Hipoteza je da metoda najbolje funkcionira na sustavima čiji se fazni prijelaz karakterizira kao prijelaz iz uređenog u neuređeno stanje i obrnuto, što bi trebalo dodatno istražiti.

Dodaci

Dodatak A Kod

A.1 Kod za generiranje podataka

generate_data.py

```
import numpy as np
from tqdm.auto import tqdm

def calc_delta_E_square(system, arr_idx):
    (N, L, L) = system.shape
    neighbours = np.zeros((N))
    for i in [-1, 1]:
        help_idx = arr_idx.copy()
        help_idx[:, 1] = help_idx[:, 1] + i
        help_idx[:, 1:] = help_idx[:, 1:] % L
        neighbours += system[tuple(help_idx.T)]
    for j in [-1, 1]:
        help_idx = arr_idx.copy()
        help_idx[:, 2] = help_idx[:, 2] + j
        help_idx[:, 1:] = help_idx[:, 1:] % L
        neighbours += system[tuple(help_idx.T)]
    delta_e = 2 * system[tuple(arr_idx.T)] * neighbours
    return delta_e

def calc_delta_e_triangular(system, arr_idx):
    (N, L, L) = system.shape
    neighbours = np.zeros((N))
    for i in [-2, 2]:
        help_idx = arr_idx.copy()
        help_idx[:, 1] = help_idx[:, 1] + i
```

```
            help_idx[:, 1:] = help_idx[:, 1:] % L
            neighbours += system[tuple(help_idx.T)]
        for i, j in [(-1, -1), (-1, 1), (1, -1), (1, 1)]:
            help_idx = arr_idx.copy()
            help_idx[:, 1] = help_idx[:, 1] + i
            help_idx[:, 2] = help_idx[:, 2] + j
            help_idx[:, 1:] = help_idx[:, 1:] % L
            neighbours += system[tuple(help_idx.T)]
        delta_E = 2 * system[tuple(arr_idx.T)] * neighbours
        return delta_E


def calc_delta_e_hexagonal(system, arr_idx):
    (N, L, L) = system.shape
    neighbours = np.zeros((N))
    for j in [-1, 1]:
        help_idx = arr_idx.copy()
        help_idx[:, 2] = help_idx[:, 2] + j
        help_idx[:, 1:] = help_idx[:, 1:] % L
        neighbours += system[tuple(help_idx.T)]
    for i, j in zip([-1, 1], [-1, 1]):
        help_idx = arr_idx.copy()
        help_idx[:, 1] = help_idx[:, 1] + i
        help_idx[:, 2] = help_idx[:, 2] + j
        help_idx[:, 1:] = help_idx[:, 1:] % L
        neighbours += system[tuple(help_idx.T)]
    for i, j in zip([-1, 1], [1, -1]):
        help_idx = arr_idx.copy()
        help_idx[:, 1] = help_idx[:, 1] + i
        help_idx[:, 2] = help_idx[:, 2] + j
        help_idx[:, 1:] = help_idx[:, 1:] % L
        neighbours += system[tuple(help_idx.T)]
    delta_E = 2 * system[tuple(arr_idx.T)] * neighbours
```

```
59      return delta_E
60
61
62  def flip(delta_e, T):
63      T = np.clip(T, 1e-3, None)
64      N = len(delta_e)
65      mask = np.ones((N))
66      mask[np.where(((delta_e <= 0) | (np.random.random(size=(N)) \
67              < np.exp(-delta_e / T))) == True)] = -1
68      return mask
69
70
71  def generate_examples(T, L, N, lattace):
72      calc_delta_E = eval("calc_delta_E_" + lattace)
73      if lattace == "hexagonal":
74          el = np.random.randint(0, 2, size=(N))
75          first = np.tile(np.array([el, el, -el + 1, -el + 1]).T, \
76                                          reps=[1, L // 4 + 1])
77          second = -first + 1
78          tile = np.stack([first, second], axis=2)
79          tile = np.swapaxes(tile, 1, 2)
80          mask = np.tile(tile, reps=[1, L // 2 + 1, 1])
81          mask_hex = mask[:, :L, :L]
82      if lattace == "triangular":
83          el = np.random.randint(0, 2, size=(N))
84          first = np.tile(np.array([el, -el + 1]).T, reps=[1, L // 2 + 1])
85          second = -first + 1
86          tile = np.stack([first, second], axis=2)
87          tile = np.swapaxes(tile, 1, 2)
88          mask = np.tile(tile, reps=[1, L // 2 + 1, 1])
89          mask_hex = mask[:, :L, :L]
90      else:
91          mask_hex = np.ones((N, L, L))
```

```
92      system = np.random.randint(0, 2, size=(N, L, L))
93      system[np.where(system == 0)] = -1
94      system = system * mask_hex
95      for it in range(100 * L * L):
96          idx = np.random.randint(0, L, size=(N, 2))
97          x = idx[:, 0]
98          y = idx[:, 1]
99          tuples = [(i, x_i, y_i) for i, (x_i, y_i) in enumerate(zip(x, y))]
100         arr_idx = np.array(tuples)
101         delta_e = calc_delta_E(system, arr_idx)
102         mask = flip(delta_e, T)
103         system[tuple(arr_idx.T)] = system[tuple(arr_idx.T)] * mask
104         system = system * mask_hex
105     return system
106
107
108 def run(lattace, num_per_temp=1000, L=10, min_temp=0, max_temp=6, step=0.25):
109     if lattace not in ["square", "triangular", "hexagonal"]:
110         raise ValueError("lattace must be one of \
111                           [square, triangular, hexagonal]")
112     configurations = []
113     temperatures = []
114     for T in tqdm(np.arange(min_temp, max_temp + step, step=step)):
115         example = generate_examples(T, L, num_per_temp, lattace)
116         configurations.append(example)
117         temperatures.append(np.ones((num_per_temp)) * T)
118     x = np.array(configurations).reshape((-1, L, L))
119     y = np.array(temperatures).reshape((-1, 1))
120     return x, y
```

A.2 Kod s modelima

models.py

```
 from torch import nn
import torch
import torch.nn.functional as F
import torch.autograd as autograd


class DenseBnRelu(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DenseBnRelu, self).__init__()
        self.dens = nn.Sequential(
                    nn.Linear(in_channels, out_channels),
                    nn.ELU(inplace = True),
                    nn.BatchNorm1d(out_channels,),
                    nn.Dropout(0.25))
    def forward(self, x):
        x = self.dens(x)
        return x

class temperature_network(nn.Module):
    def __init__(self,in_featuers):
        super().__init__()
        self.hidden = DenseBnRelu(in_featuers,64)
        self.hidden2 = DenseBnRelu(64,64)
        self.output = nn.Linear(64, 25)
    def forward(self, x):
        x = self.hidden(x)
        x = self.hidden2(x)
        x = self.output(x)
        return x

```

```
31  class RBM(nn.Module):
32      def __init__(self, nv=28*28, nh=512, cd_steps=1):
33          super(RBM, self).__init__()
34          self.W = nn.Parameter(torch.randn(nv, nh) * 0.01)
35          self.bv = nn.Parameter(torch.zeros(nv))
36          self.bh = nn.Parameter(torch.zeros(nh))
37          self.cd_steps = cd_steps
38
39      def bernoulli(self, p):
40          return torch.sign(p -  \
41                  autograd.Variable(2*torch.rand(p.size())-1).cuda())
42      def energy(self, v):
43          b_term = v.mv(self.bv)
44          linear_tranform = F.linear(v, self.W.t(), self.bh)
45          h_term = linear_tranform.exp().add(1).log().sum(1)
46          return (-h_term -b_term).mean()
47
48      def sample_h(self, v):
49          ph_given_v = torch.tanh(F.linear(v, self.W.t(), self.bh))
50          return self.bernoulli(ph_given_v)
51      def sample_v(self, h):
52          pv_given_h = torch.tanh(F.linear(h, self.W, self.bv))
53          return self.bernoulli(pv_given_h)
54      def forward(self, v):
55          vk = v.clone()
56          for step in range(self.cd_steps):
57              hk = self.sample_h(vk)
58              vk = self.sample_v(hk)
59              vk = vk*v.abs()
60          return v, vk.detach()
61
62  class VAE(nn.Module):
63      def __init__(self, input_dim):
```

```
64          super(VAE, self).__init__()
65          self.fc1 = nn.Linear(input_dim,128)
66          self.fc21 = nn.Linear(128, 64)
67          self.fc22 = nn.Linear(128, 64)
68          self.fc3 = nn.Linear(64,128)
69          self.fc4 = nn.Linear(128, input_dim)
70      def encode(self, x):
71          x = nn.ELU()(self.fc1(x))
72          return self.fc21(x), self.fc22(x)
73      def reparameterize(self, mu, logvar):
74          std = torch.exp(0.5*logvar)
75          eps = torch.randn_like(std)
76          return mu + eps*std
77      def decode(self, x):
78          x = nn.ELU()(self.fc3(x))
79          return torch.tanh(self.fc4(x))
80      def forward(self, x):
81          mu, logvar = self.encode(x)
82          z = self.reparameterize(mu, logvar)
83          out = self.decode(z)
84          out = out*x.abs()
85          return out , mu, logvar
86
87  class Generator(nn.Module):
88      def __init__(self, input_length: int):
89          super(Generator, self).__init__()
90          self.dense_layer = nn.Linear(int(input_length), 128)
91          self.dense_layer1_1 = nn.Linear(128, 64)
92          self.dense_layer1_2 = nn.Linear(64, 128)
93          self.dense_layer2 = nn.Linear(128,int(input_length))
94          self.activation = nn.Sigmoid()
95          self.activation2 = nn.Tanh()
96          self.drop = nn.Dropout(0)
```

```
97      def forward(self, x):
98          hex_mask = x.abs()
99          x = self.activation(self.dense_layer(x))
100         x = self.drop(x)
101         x = self.activation(self.dense_layer1_1(x))
102         x = self.activation(self.dense_layer1_2(x))
103         x = self.activation2(self.dense_layer2(x))
104         x = x * hex_mask
105         return x
106
107 class Discriminator(nn.Module):
108     def __init__(self, input_length: int):
109         super(Discriminator, self).__init__()
110         self.dense = nn.Linear(int(input_length), 64);
111         self.dense2 = nn.Linear(64, 1);
112         self.activation = nn.Sigmoid()
113         self.activation2 = nn.Sigmoid()
114     def forward(self, x):
115         x = self.activation(self.dense(x))
116         return self.activation2(self.dense2(x))
```

A.3 Kod s pomoćnim funkcijama

utils.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pickle
4 import torch
5 from torch import nn
6 import torch.autograd as autograd
7 from torch.nn import functional as F
8 from torch.utils.data import DataLoader, Dataset as torc_dataset
```

```
9
10  class Dataset(torc_dataset):
11      def __init__(self, x, y):
12          y = y.reshape((-1))
13          self.x = torch.from_numpy(x)
14          self.y = torch.from_numpy(y)
15          self.dataset_len = len(x)
16      def __getitem__(self, idx):
17          x_example = self.x[idx]
18          y_example = self.y[idx]
19          return x_example.float(), y_example
20      def __len__(self):
21          return self.dataset_len
22
23  def get_dataloader(x, y, batch_size=512, shuffle=True, sampler=None):
24      dataset = Dataset(x, y)
25      loader = DataLoader(dataset=dataset,
26                          batch_size=batch_size,
27                          shuffle=shuffle,
28                          sampler=sampler,
29                          pin_memory=True)
30      return loader
31
32  def epoch(data_loader, model, loss_fn, optimizer=None, train=True):
33      if train:
34          model.train()
35      else:
36          model.eval()
37      loss_tot = []
38      for x, y in data_loader:
39          x = x.cuda()
40          y = y.cuda()  # *0.25
41          prediction = model(x)
```

```
            loss = loss_fn(prediction, y)
            if train:
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()
            loss_tot.append(loss.cpu().detach().numpy())
        return np.mean(loss_tot), model

def plot_losses(train_loss, valid_loss, name, label1="Train", label2="Valid"):
    plt.rc('xtick', labelsize=20)
    plt.rc('ytick', labelsize=20)
    plt.rc('axes', labelsize=25)
    plt.figure(figsize=(7, 5))
    plt.plot(train_loss, label=label1)
    plt.plot(valid_loss, label=label2)
    plt.legend(prop={'size': 22})
    plt.xlabel('Iteracija')
    plt.ylabel('Greška')
    plt.savefig(name, bbox_inches="tight")
    plt.show()

def get_data_loaders(file, L, batch_size, shape, use_for_flow=False):
    if shape not in ["1D", "2D"]:
        raise ValueError("shape must be 1D or 2D")
    if shape == "2D":
        tuple_shape = (-1, 1, L, L)
    if shape == "1D":
        tuple_shape = (-1, L ** 2)
    data_dict = pickle.load(open(file, "rb"))
    x_train = data_dict["x_train"].astype(np.float32)
    y_train = data_dict["y_train"].astype(np.float32)
    x_test = data_dict["x_test"].astype(np.float32)
    y_test = data_dict["y_test"].astype(np.float32)
```

```
75      x_train = x_train.reshape(tuple_shape)
76      x_test = x_test.reshape(tuple_shape)
77      y_train = y_train / 0.25
78      y_test = y_test / 0.25
79      y_train = np.round(y_train)
80      y_test = np.round(y_test)
81      num_of_temps = max(y_train)[0] + 1
82      num_of_temps = int(num_of_temps)
83      batch = 512
84      train_loader = get_dataloader(x_train, y_train.astype(int), \
85                                    batch, shuffle=True)
86      valid_loader = get_dataloader(x_test, y_test.astype(int), \
87                                    batch, shuffle=False)
88      if use_for_flow:
89          return x_test, y_test
90      return train_loader, valid_loader, num_of_temps
91
92  def rbm_epoch(data_loader, model, optimizer=None, train=True):
93      if train:
94          model.train()
95      else:
96          model.eval()
97      loss_tot = []
98      for x, y in data_loader:
99          x_batch = autograd.Variable(x.cuda())
100         v, vk = model(x_batch)
101         loss = model.energy(v) - model.energy(vk)
102         if train:
103             loss.backward()
104             optimizer.step()
105             optimizer.zero_grad()
106         plt_loss = np.mean((v - vk).cpu().detach().numpy() ** 2)
107         loss_tot.append(plt_loss)
```

```
108         return np.mean(loss_tot), model
109
110     def loss_function_vae(x, output):
111         recon_x, mu, logvar = output
112         recon_x = (recon_x + 1) / 2
113         x = (x + 1) / 2
114         BCE = F.binary_cross_entropy(recon_x, x, reduction='mean')
115         KLD = -0.5 * torch.mean(1 + logvar - mu.pow(2) - logvar.exp())
116         return BCE + KLD
117
118     def vae_epoch(data_loader, model, optimizer=None, train=True):
119         if train:
120             model.train()
121         else:
122             model.eval()
123         loss_tot = []
124         for x, y in data_loader:
125             x = x.cuda()
126             y = y.cuda()
127             output = model(x)
128             loss = loss_function_vae(x, output)
129             if train:
130                 loss.backward()
131                 optimizer.step()
132                 optimizer.zero_grad()
133             loss_tot.append(loss.cpu().detach().numpy())
134         return np.mean(loss_tot), model
135
136     def gan_epoch(data_loader, generator, discriminator,  \
137                     generator_optimizer, discriminator_optimizer):
138         gen_loss_tot = []
139         dis_loss_tot = []
140         loss = nn.BCELoss()
```

```
141         for x, y in data_loader:
142             generator_optimizer.zero_grad()
143             true_data = x
144             true_labels = [1] * len(x)
145             true_data = true_data.float().cuda()
146             generated_data = generator(true_data)
147             generator_loss = 0
148             true_labels = torch.tensor(true_labels).float().cuda()
149             generator_discriminator_out = discriminator(generated_data)
150 
151             generator_loss += loss(generator_discriminator_out, \
152                                    true_labels.reshape((-1, 1)))
153             generator_loss += torch.mean((generated_data.mean(1) - \
154                                        true_data.mean(1)) ** 2)
155             generator_loss.backward()
156             if True:
157                 generator_optimizer.step()
158             discriminator_optimizer.zero_grad()
159             true_discriminator_out = discriminator(true_data)
160             true_discriminator_loss = loss(true_discriminator_out, true_labels)
161             generator_discriminator_out = discriminator(generated_data.detach())
162             generator_discriminator_loss = loss(generator_discriminator_out, \
163                                             torch.zeros(len(x)).cuda().reshape((-1, 1
164             discriminator_loss = (true_discriminator_loss + \
165                                    generator_discriminator_loss) / 2
166             discriminator_loss.backward()
167             if np.random.uniform() < 0.2:
168                 discriminator_optimizer.step()
169             gen_loss_tot.append(generator_loss.cpu().detach().numpy())
170             dis_loss_tot.append(discriminator_loss.cpu().detach().numpy())
171         return np.mean(gen_loss_tot), np.mean(dis_loss_tot),  \
172                 generator, discriminator
173 
```

```
174 def sample_prob(next_state_probs, L):
175     rands = torch.rand(len(next_state_probs), L ** 2).cuda() * 2 - 1
176     spins = (next_state_probs >= rands).float() * 2 - 1
177     spins[torch.where(next_state_probs == 0)] = 0
178     next_state = spins
179     return next_state
180 
181 def rg_flow(model_class, generator, starting_temp, generator_name, L, x_test):
182     n = 50
183     num_iter = 10000
184     idx = np.random.randint(1000, size=(n))
185     start_idx = int(starting_temp / 0.25) * 1000
186     temperature_dist = []
187     start = x_test[idx + start_idx].reshape((-1, L ** 2)).astype(np.float32)
188     next_state = start.copy()
189     next_state = torch.from_numpy(next_state).cuda()
190     for i in range(num_iter):
191         current_dist = model_class(next_state).softmax(dim=1)
192         temperature_dist.append( \
193                 current_dist.mean(0).cpu().detach().numpy().reshape((-1)))
194         if generator_name == "rbm":
195             _, next_state = generator(next_state)
196         elif generator_name == "vae":
197             next_state_probs, _, _ = generator(next_state)
198             next_state = sample_prob(next_state_probs, L)
199         elif generator_name == "gan":
200             next_state_probs = generator(next_state)
201             next_state = sample_prob(next_state_probs, L)
202         next_state = next_state.detach()
203     rez = np.array(temperature_dist)
204     max_p = np.argmax(rez, 1) * 0.25
205     converged_temp = np.mean(max_p[-1000:])
206     return converged_temp
```

```
207
208 def rg_flow_without_gen(model_class, starting_temp, L, x_test):
209     n = 50
210     num_iter = 20000
211     idx = np.random.randint(1000, size=(n))
212     start_idx = int(starting_temp / 0.25) * 1000
213     temperature_dist = []
214     start = x_test[idx + start_idx].reshape((-1, L ** 2)).astype(np.float32)
215     next_state = start.copy()
216     next_state = torch.from_numpy(next_state).cuda()
217     for i in range(num_iter):
218         current_dist = model_class(next_state).softmax(dim=1)
219         temperature_dist.append( \
220                 current_dist.mean(0).cpu().detach().numpy().reshape((-1)))
221         next_state = next_state.detach()
222     rez = np.array(temperature_dist)
223     max_p = np.argmax(rez, 1) * 0.25
224     converged_temp = np.mean(max_p[-10000:])
225     return converged_temp
```

A.4 Kod za učenje i generiranje

main.py

```
1 import pickle
2 import generate_data
3 L = 20
4 batch_size = 512
5 ########################
6 ### GENERATING DATA ###
7 ########################
8 for lattace in ["square", "triangular", "hexagonal"]:
9     x_train, y_train = generate_data.run(lattace, L=L)
```

```
10      x_test, y_test = generate_data.run(lattace, L=L)
11      data_dict = {}
12      data_dict["x_train"] = x_train
13      data_dict["y_train"] = y_train
14      data_dict["x_test"] = x_test
15      data_dict["y_test"] = y_test
16      pickle.dump(data_dict, open("./data_20_{}.pk".format(lattace), "wb"))
17  ##########################
18  ### TEMPERATURE MODEL ###
19  ##########################
20  from models import temperature_network
21  from utils import *
22  from tqdm.auto import tqdm
23  temp_models = []
24  for lattace in ["square", "triangular", "hexagonal"]:
25      max_epoch = 200
26      train_loader, valid_loader, num_of_temps = get_data_loaders( \
27                      "./data_20_{}.pk".format(lattace), L, batch_size, "1D")
28      model_class = temperature_network(L ** 2).cuda()
29      loss_fn = nn.CrossEntropyLoss()
30      p = tqdm(range(max_epoch))
31      optimizer = torch.optim.Adam(model_class.parameters(), lr=0.0003)
32      model_class.train()
33      train_loss = []
34      valid_loss = []
35      for i in p:
36          loss, model_class = epoch(train_loader, model_class, loss_fn, \
37                                    optimizer, train=True)
38          train_loss.append(loss)
39          loss, model_class = epoch(valid_loader, model_class, \
40                                    loss_fn, train=False)
41          valid_loss.append(loss)
42          p.set_description("train: {0:.4}".format(train_loss[-1]) + \
```

```
                    " valid: " + "{0:.4}".format(valid_loss[-1]))
    temp_models.append(model_class)
    plot_losses(train_loss, valid_loss,  \
            "./graphs/temperature_network_{}.jpg".format(lattace))
###########
### RBM ###
###########
from models import RBM
rbm_models = []
for lattace in ["square", "triangular", "hexagonal"]:
    train_loader, valid_loader, num_of_temps = get_data_loaders( \
            "./data_20_{}.pk".format(lattace), L, batch_size, "1D")
    rbm = RBM(L ** 2, 64).cuda()
    optimizer = torch.optim.Adam(rbm.parameters(), 0.0001)
    epochs = 200
    p_bar = tqdm(range(epochs))
    train_loss = []
    valid_loss = []
    for i in p_bar:
        loss, rbm = rbm_epoch(train_loader, rbm, optimizer, train=True)
        train_loss.append(loss)
        loss, rbm = rbm_epoch(valid_loader, rbm, train=False)
        valid_loss.append(loss)
    rbm_models.append(rbm)
    plot_losses(train_loss, valid_loss, "./graphs/rbm_{}.jpg".format(lattace))
    W = rbm.W.cpu().detach().numpy()
    plt.imshow(np.matmul(W, W.T))
    plt.show()
###########
### VAE ###
###########
from models import VAE
vae_models = []
```

```
for lattace in ["square", "triangular", "hexagonal"]:
    train_loader, valid_loader, num_of_temps = get_data_loaders(\
            "./data_20_{}.pk".format(lattace), L, batch_size, "1D")
    vae_model = VAE(L ** 2).cuda()
    optimizer = torch.optim.Adamax(vae_model.parameters(), lr=1e-4)
    epochs = 100
    p_bar = tqdm(range(epochs))
    train_loss = []
    valid_loss = []
    for i in p_bar:
        loss, vae_model = vae_epoch(train_loader, vae_model, \
                                    optimizer, train=True)
        train_loss.append(loss)
        loss, vae_model = vae_epoch(valid_loader, vae_model, train=False)
        valid_loss.append(loss)
        p_bar.set_description("train: {0:.4}".format(train_loss[-1]) + \
                        " valid: " + "{0:.4}".format(valid_loss[-1]))
    vae_models.append(vae_model)
    plot_losses(train_loss, valid_loss, "./graphs/vae_{}.jpg".format(lattace))
###########
### GAN ###
###########
from models import Generator, Discriminator
gan_models = []
for lattace in ["square", "triangular", "hexagonal"]:
    train_loader, valid_loader, num_of_temps = get_data_loaders( \
                "./data_20_{}.pk".format(lattace), L, batch_size, "1D")
    generator = Generator(L ** 2).cuda()
    discriminator = Discriminator(L ** 2).cuda()
    generator_optimizer = torch.optim.Adam(generator.parameters(), lr=0.0001)
    discriminator_optimizer = torch.optim.Adam(discriminator.parameters(),\
                                               lr=0.00003)
    epochs = 100
```

```
    p_bar = tqdm(range(epochs))
    gen_loss = []
    dis_loss = []
    for i in p_bar:
        gen_l, dis_l, generator, discriminator = gan_epoch(train_loader,\
                            generator, discriminator, generator_optimizer, \
                            discriminator_optimizer)
        gen_loss.append(gen_l)
        dis_loss.append(dis_l)
        p_bar.set_description(
            "Generator: {0:.4}".format(gen_loss[-1]) + " Diskriminator: " + \
                                    "{0:.4}".format(dis_loss[-1]))
    gan_models.append(generator)
    plot_losses(gen_loss, dis_loss, "./graphs/gan_d_{}.jpg".format(lattace), \
                                            "Generator", "Diskriminator")
###############
### RG FLOW ###
###############
import pandas as pd
results = []
p_bar = tqdm(total=3 * 3 * 7)
for lattace_idx, lattace in enumerate(["square", "triangular", "hexagonal"]):
    x_test, y_test = get_data_loaders("./data_20_{}.pk".format(lattace), \
                                L, batch_size, "1D", use_for_flow=True)
    for generator_name in ["rbm", "vae", "gan"]:
        if generator_name == "rbm":
            generator = rbm_models[lattace_idx]
        if generator_name == "vae":
            generator = vae_models[lattace_idx]
        if generator_name == "gan":
            generator = gan_models[lattace_idx]
        model_class = temp_models[lattace_idx]
        generator = generator.eval()
```

```
        model_class = model_class.eval()
        for starting_temp in [0, 1, 2, 3, 4, 5, 6]:
            final_temp = rg_flow(model_class, generator, \
                    starting_temp, generator_name, L, x_test)
            results.append([lattace, generator_name, starting_temp, final_temp])
            p_bar.update(1)
results_df = pd.DataFrame.from_records(results, columns=["LATTACE", \
                    "GENERATOR", "STARTING_TEMP", "FINAL_TEMP"])
pivoted = results_df.pivot_table(values="FINAL_TEMP", \
                            index=["GENERATOR", "LATTACE"], \
pivoted.columns.name = None
custom_dict_2 = {"square": 0, "triangular": 1, "hexagonal": 2}
pivoted["sort"] = pivoted.apply(lambda x: custom_dict_2[x["LATTACE"]], axis=1)
pivoted.sort_values(by="sort", inplace=True)
pivoted.drop(columns=["sort"], inplace=True)
pivoted["AVG_FINAL_TEMP"] = pivoted[[i for i in range(7)]].mean(1)
true_temp = {"square": 2.269, "triangular": 3.641, "hexagonal": 1.519}
pivoted["TRUE_CRITICAL_TEMP"] = pivoted["LATTACE"].apply(lambda x: true_temp[x])
pivoted.to_csv("./results_pivoted.csv", index=False)
##################################
### RG FLOW WITHOUT GENERATOR ###
##################################
import copy
results2 = []
p_bar = tqdm(total=3 * 7)
for lattace_idx, lattace in enumerate(["square", "triangular", "hexagonal"]):
    x_test, y_test = get_data_loaders("./data_20_{}.pk".format(lattace), \
                        L, batch_size, "1D", use_for_flow=True)
    for starting_temp in [0, 1, 2, 3, 4, 5, 6]:
        model_class = copy.deepcopy(temp_models[lattace_idx])
        model_class = model_class.train()
        final_temp = rg_flow_without_gen(model_class, starting_temp, L, x_test)
        results2.append([lattace, starting_temp, final_temp])
```

```
175             p_bar.update(1)
176     results_df = pd.DataFrame.from_records(results2, \
177             columns=["LATTACE", "STARTING_TEMP", "FINAL_TEMP"])
178     pivoted = results_df.pivot_table(values="FINAL_TEMP", index=["LATTACE"], \
179                                      columns=["STARTING_TEMP"]).reset_index()
180     pivoted.columns.name = None
181     custom_dict_2 = {"square": 0, "triangular": 1, "hexagonal": 2}
182     pivoted["sort"] = pivoted.apply(lambda x: custom_dict_2[x["LATTACE"]], axis=1)
183     pivoted.sort_values(by="sort", inplace=True)
184     pivoted.drop(columns=["sort"], inplace=True)
185     pivoted["AVG_FINAL_TEMP"] = pivoted[[i for i in range(7)]].mean(1)
186     true_temp = {"square": 2.269, "triangular": 3.641, "hexagonal": 1.519}
187     pivoted["TRUE_CRITICAL_TEMP"] = pivoted["LATTACE"].apply(lambda x: true_temp[x])
188     pivoted.to_csv("./res_without_gen.csv", index=False, )
```

Literatura

[1] Rajatm R.; Madhavan, A.; Ng, A. Large-scale Deep Unsupervised Learning using Graphics Processors. // ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning

[2] Kittel, C. Introduction to Solid State Physics. 8th ed. John Wiley & Sons, Inc., 2005.

[3] Fendley, P. Modern Statistical Mechanics. 1st ed. The University of Virginia, 2014

[4] Fizika čvrstog stanja 1 & 2, Fazni prijelazi, http://grdelin.phy.hr/~ivo/Nastava/Fizika_Cvrstog_Stanja/index.php, 10.7.2020.

[5] Correlation function (statistical mechanics), https://en.wikipedia.org/wiki/Correlation_function_(statistical_mechanics), 10.7.2020.

[6] Kadanoff, L. P. Scaling laws for Ising models near T_c. // Physics Physique Fizika. 2 (1966), str. 263.

[7] Laguna, J. R. Rgkadanoff.png, https://commons.wikimedia.org/wiki/File:Rgkadanoff.png, 11.7.2020.

[8] Renormalization group, Block spin, https://en.wikipedia.org/wiki/Renormalization_group, 11.7.2020.

[9] Gallavotti, G. Statistical Mechanics: A Short Treatise (Theoretical and Mathematical Physics). Berlin: Springer-Verlag, 1999 ed.

[10] Martin Evans. Mean-Field Theory of the Ising Model, https://www2.ph.ed.ac.uk/~mevans/sp/sp10.pdf, 11.7.2020.

[11] Google Neural Machine Translation, https://en.wikipedia.org/wiki/Google_Neural_Machine_Translation, 15.7.2020.

[12] Mehta, P.; Bukov, M.; Wang, C.; Day, A.; Richardson, C.; Fisher, C.; Schwab, D. A high-bias, low-variance introduction to Machine Learning for physicists. // Physics Reports, Vol. 210 (2018)

[13] Rumelhart, D. E.; Hinton, G. E.; Williams, R. J. Learning representations by back-propagating errors. // Nature, Vol. 323 (1986), str. 533-536.

[14] Zeiler M. D.; Fergus R. Visualizing and Understanding Convolutional Networks. // Computer Vision – ECCV 2014.

[15] Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. // The Journal of Machine Learning Research, Vol. 15 (2014).

[16] Gradient descent, https://en.wikipedia.org/wiki/Gradient_descent, 18.7.2020.

[17] Kingma, D.; Ba, J. Adam: A Method for Stochastic Optimization. // International Conference on Learning Representations (2014)

[18] Zeiler, M. ADADELTA: An adaptive learning rate method. // ArXiv, Vol. abs/1212.5701 (2012).

[19] Alexandrov, A. Gradient descent, https://commons.wikimedia.org/wiki/File:Gradient_descent.png, 18.7.2020.

[20] Restricted Boltzmann machine, https://commons.wikimedia.org/wiki/File:Restricted_Boltzmann_machine.svg, 18.7.2020.

[21] Hinton, G. A Practical Guide to Training Restricted Boltzmann Machines. // Technical Report UTML TR 2010-003, University of Toronto (2010)

[22] Pearson, K On lines and planes of closest fit to systems of points in space. // Philosophical Magazine, Vol. 2 (1901), str. 559-572.

[23] Image Compression Using Autoencoders in Keras, https://blog.paperspace.com/autoencoder-image-compression-keras/, 19.7.2020.

[24] Kullback, S.; Leibler, R.A.On information and sufficiency. // Annals of Mathematical Statistics, Vol. 22(1) (1951), str. 79–86.

[25] Goodfellow, J.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio Y. Generative adversarial nets. // NIPS'14: Proceedings of the 27th International Conference on Neural Information Processing Systems 2014

[26] Learning Generative Adversarial Networks (GANs), https://mc.ai/learning-generative-adversarial-networks-gans/, 19.7.2020.

[27] Ellen de Mello Koch, E.M.; Koch, R.M.; Cheng, L. Is Deep Learning a Renormalization Group Flow?. // ArXiv, Vol. abs/1906.05212 (2019).

[28] Iso, S.; Shiba, S.; Yokoo, S. Scale-invariant feature extraction of neural network and renormalization group flow. //Phys. Rev. , Vol. 97 (2018).

[29] Beny, C. Deep learning and the renormalization group. // ArXiv, Vol. abs/1301.3124 (2013).

[30] Python, `https://www.python.org/`, 25.7.2020.

[31] NumPy, `https://numpy.org/`, 25.7.2020.

[32] PyTorch, `https://pytorch.org/`, 25.7.2020.

[33] Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. // ArXiv, Vol. abs/1502.03167 (2015).