Predviđanje ishoda šahovske igre korištenjem rekurentnih neuronskih mreža

Vuković, Filip

Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni diplomski studij matematike
smjer: Matematika i računarstvo

Predviđanje ishoda šahovske igre korištenjem rekurentnih neuronskih mreža

Diplomski rad

Mentor:

izv.prof.dr.sc.
Domagoj Matijević

Kandidat:

Filip Vuković

Osijek, 2023

Sadržaj

Uvod

Šah je jedna od najpoznatijih i najkompleksnijih igara današnjice. Šahovska teorija razvijala se stoljećima, ali najveći napredak došao je razvojem računalnih sustava za igranje šaha u zadnjih 30 godina. Prekretnica u ovom razvoju dogodila se 1997. godine kada je IBM-ov Deep Blue pobijedio tada šahovskog prvaka Gary Kasparova. To je bila prva pobjeda računala nad šahovskim prvakom te je značajan događaj u razvoju umjetne inteligencije. Danas, računalni sustavi za igranje šaha poput Stockfisha, Alpha Zero-a i Leele temelje se na neuronskim mrežama te su daleko nadmašile ljudsko znanje i vještinu. Primjerice, uzmimo u obzir sustav mjerenja jačine šahovskih igrača ELO koji se danas koristi za rangiranje igrača. Najveći postignuti ljudski ELO 2882 ostvario je Magnus Carlsen 2014. godine dok je Stockfish-ev procjenjeni ELO trenutno 3532, što predstavlja značajnu razliku u vještini igranja igre. Iako su vještinom igre prestigli ljude, ovi modeli i dalje se poboljšavaju, a najčešća uporaba im je razvoj šahovske teorije i analiza igara u svrhu poboljšanja ljudske igre.

Svi prethodno navedeni sustavi temelje se na analizi dane šahovske pozicije i odabiru najboljeg idućeg poteza. U ovom radu promotrit ćemo drugačiji pristup: predstavljanje šahovske igre kao niza poteza. Ovaj pristup nije čest među neuronskim modelima za šah iako je gledanje šahovske igre kao niz poteza vrlo intuitivno. Razlog tome je što ovakvi modeli nisu pogodni za igranje šaha nego su primjereniji za probleme poput predviđavanja ishoda šahovske igre.

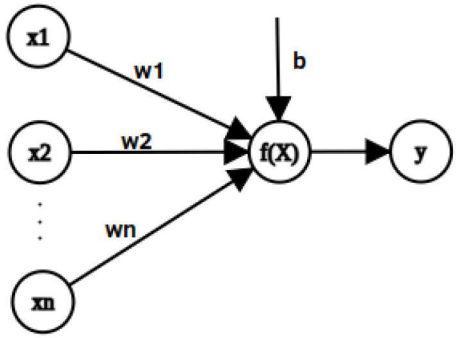
Cilj ovog rada je konstruirati neuronski model koji će pomoću danog niza poteza u standardnoj šahovskoj notaciji odrediti rezultat šahovske igre: pobjeda bijelog, remi ili pobjeda crnog. Isto tako, time ćemo pokazati može li se šahovska igra obrađivati kao sekvenca i pogledat ćemo utjecaj različitih reprezentacija ploče na rezultate te ih usporediti.
Kroz rad, prvo ćemo se upoznati s teorijom potrebnom za razumijevanje neuronskih mreža i tehnikama korištenima pri treniranju modela.
U drugom poglavlju nastavit ćemo s teorijom i obraditi rekurentne neuronske mreže i LSTM arhitekturu potrebnu za konstrukciju modela.
U trećem poglavlju pogledat ćemo razlike u različitim reprezentacijama šahovske ploče, opisati ćemo strukturu modela i metode korištene za treniranje i testiranje modela. Na kraju, pogledat ćemo rezultate koje naš model ostvaruje i usporedbu rezultata s obzirom na različite parametre modela.

1 | Osnove neuronskih mreža

Struktura i način rada neuronskih mreža inspirirana je radom ljudskog mozga. Ideja korištenja neuronskih mreža u računalne svrhe potječe iz 1943. godine autora McCulloch i Pitts, dok je prvi prijedlog za mrežu koja se može trenirati dao Rosenblatt 1957. godine i nazvao je *perceptron*. Usprkos korijenima u 40-tim i 50-tim godinama prošlog stoljeća, ova grana računalne znanosti najveći napredak postiže u 21. stoljeću. U ovom poglavlju definirat ćemo neuron i najjednostavniji oblik neuronskih mreža te pokazati princip rada i treniranja.

1.1 Perceptron model i višeslojne neuronske mreže

Struktura perceptron modela inspirirana je građom neurona u mozgu. Sastoji se od jednog ulaznog sloja i jednog izlaznog čvora. Za dani skup ulaznih podataka $n \in \mathbb{N}$ s njihovim težina $w_1, \ldots, w_n \in \mathbb{R}$, neuron je funkcija koja računa težinsku sumu ovih ulaza i na nju primjenjuje aktivacijsku funkciju.



Slika 1.1: Osnovni model perceptrona.

Na slici 1.1 prikazan je model s ulaznim podatcima $x_1, x_2, \ldots x_n \in \mathbb{R}$ i težinama $w_1, w_2, \ldots w_n \in \mathbb{R}$. Dodatni ulaz b, zvan slobodni koeficijent (*eng. bias*), ima uvijek vrijednost 1. U nastavku ćemo s b označavati težine koji povezuju *bias* neuron s ostatkom mreže. Neuron u ovom modelu daje izlaznu vrijednost:

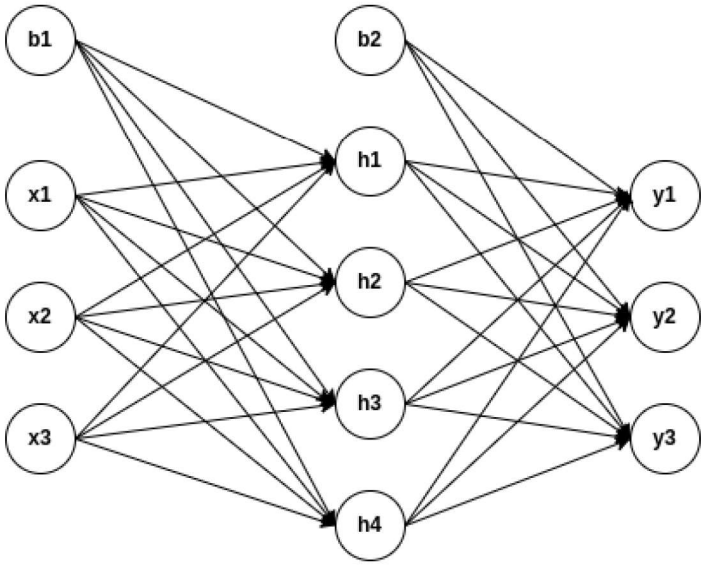
$$y := f(\sum_{i=1}^{n} w_i x_i + b) \tag{1.1}$$

gdje je $f : \mathbb{R} \to \mathbb{R}$ aktivacijska funkcija.

Korištenje jednog perceptrona za učenje složenijih problema nije moguće zbog njegove jednostavne strukture, ali uporabom više neurona i njihovim povezivanjem dolazimo do osnovne višeslojne neuronske mreže. Dok se perceptron sastoji od jednog ulaznog i izlaznog sloja u kojem se vrše sve računalne operacije, u višeslojnoj neuronskoj mreži dodaju se slojevi neurona između ulaznog i izlaznog sloja zvani skriveni slojevi (*eng.hidden layers*).

1.1.1 Feed forward neuronske mreže

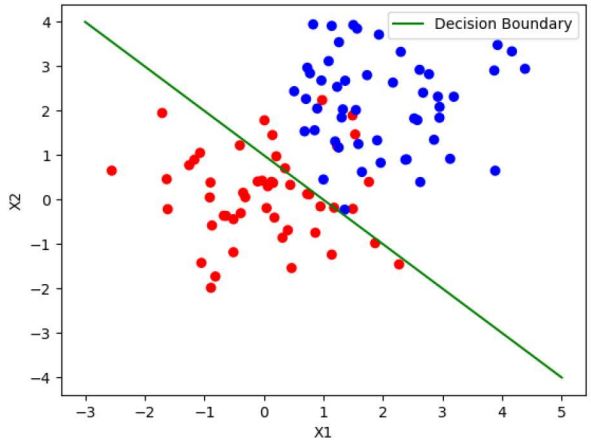
Jedna od prvih i jednostavnijih višeslojnih mreža zove se *feed-forward* mreža. Naziv je dobila po tome što se izlaz svakog sloja šalje kao ulazni podatak idućem sloju, krenuvši od ulaznog sloja do izlaznog tj. tijek računskih operacija ide u jednom smjeru - prema naprijed. Osnovna struktura ovakvih mreža podrazumijeva da su svi neuroni u jednom sloju povezani s idućim. Općenito, svojstvo povezanosti ne mora nužno biti prisutno pri modeliranju neuronskih mreža.



Slika 1.2: Struktura *feed forward* mreže

Feed forward neuronske mreže mogu se definirati i kao **usmjereni, aciklički i povezani** graf. Svojstvo acikličnosti govori nam o nedostaku petlji unutar takvih mreža te je jedna od osnovnih razlika u odnosu na mreže koje sadrže petlje (*rekurentne neuronske mreže*), koje ćemo karakterizirati u poglavlju 2. *Feed forward* mreže često se koriste u klasifikacijskim i regresijskim problemima.

Primjer 1. *Pretpostavimo da želimo rješiti problem binarne klasifikacije. Za dani skup točaka na slici 1.3 želimo napraviti model koji će moći svrstati točke u dvije klase - crvenu i plavu.*



Slika 1.3:

Cilj modela je naučiti iz danih točaka gdje se nalazi granica koja odvaja dvije klase, označena na slici sa zelenom linijom, te nove točke uspješno svrstati u jednu od dvije klase.

Za aktivacijsku funkiju u ovom primjeru prikladno je odabrati *sigmoid*, definiranu s $\sigma(x) = \frac{1}{1+e^{-x}}, \sigma \in \mathbb{R} \to (0,1)$. Za dane ulazne podatke x, pretpostavimo da smo istrenirali mrežu na skupu točaka sa slike 1.3. Treniranje je postupak prilagođavanja težina i parametara modela kako bi dobili željeni izlaz mreže. Više o treniranju mreža govorit ćemo u podpoglavlju 1.2.
Radi lakšeg zapisivanja težina u *feed forward* mrežama, koristit ćemo matrični zapis tj. matricu težina koja sadrži težine između neurona prethodnog i idućeg sloja. Dimenzije ovih matrica odgovarat će broju neurona u prethodnom i idućem sloju mreže. Predviđena vjerojatnost da x pripada klasi plavih ili crvenih točaka tada možemo izračunati s:

$$y = \sigma(W_2^T \sigma(W_1^T x + b_1) + b_2)$$

gdje je:

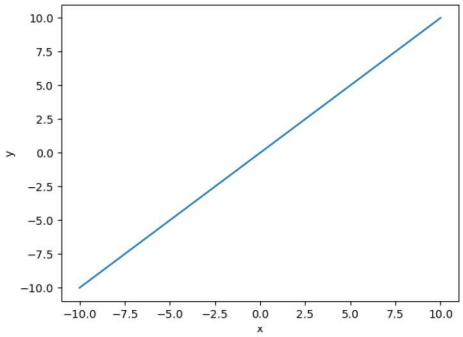
- W_1 matrica težina koja povezuje ulazne podatke i prvi, skriveni sloj
- W_2 matrica težina koja povezuje skriveni sloj i izlazni sloj
- b_1, b_2 vektori težina koje povezuju bias neurone sa skrivenim i izlaznim slojem
- $W_1^T x + b_1$ težinska suma skrivenog sloja
- $W_2^T x + b_2$ težinska suma izlaznog sloja

Izlaz ove mreže y je realna vrijednost između 0 i 1 te zaokruživanjem na najbližu cijelobrojnu vrijednost dobijemo predviđenu pripadnost klasi. Primjerice, ako mreža računa vjerojatnost da točka pripada klasi plavih točaka, vrijednosti iznad 0.5 svrstati će danu točku u plave točke, a ispod u crvene točke.

1.1.2 Aktivacijske funkcije

U primjeru 1 za izbor aktivacijske funkcije smo odabrali *sigmoid* funkciju jer je bila pogodna za postavljeni zadatak. Odabir aktivacijske funkcije koja se primjenjuje na težinsku sumu neurona iz prethodnih slojeva u mreži bitan je dio konstrukcije neuronskih mreža. Tipične aktivacijske funkcije korištene u neuronskim mrežama su:

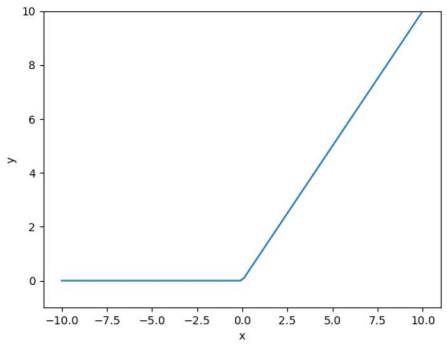
- **Identiteta**: $id(x) = x$
  Ova funkcija koristi se gotovo isključivo na ulaznom sloju te nije često korištena drugdje jer bi izlaz sloja u tom slučaju bio samo afina transformacija $W_1^T x + b$.



Slika 1.4: Graf funkcije identiteta

- **ReLU**: $f(x) = max(0, x)$

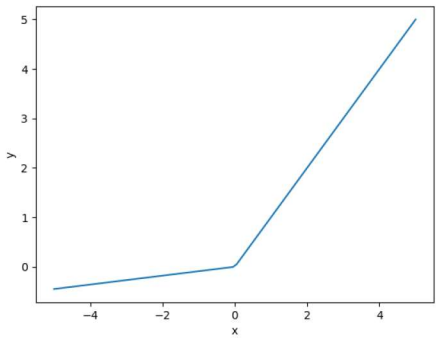
  Rectified linear function ili ReLU je po dijelovima linearna funkcija koja nije diferencijabilna, ali njezina derivacija postoji u svim točkama osim 0. Ona zamijenjuje negativne vrijednosti s nulom, a pozitivne vrijednosti ostavlja nepromijenjene. Često je korištena zbog svoje jednostavnosti i brzine. Jedan od većih nedostataka korištenja ove funkcije je loše rukovanje negativnim vrijednostima na ulazu, što može dovesti do neaktivnog stanja neurona u mreži. Primjerice, ako ReLU neuron tijekom treniranja mreže stalno dobija negativne vrijednosti na ulazu, može se dogoditi da njegov izlaz postane 0 za bilo koje ulaze te time prestane doprinositi izlazu mreže.



Slika 1.5: Graf ReLU funkcije

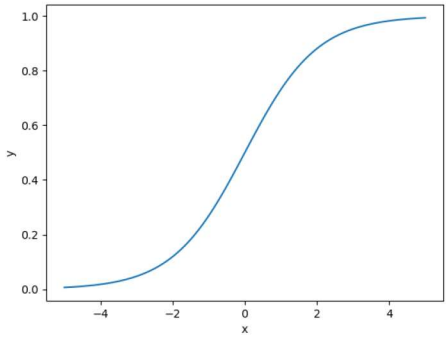
- **Leaky ReLU**: $f(x) = max(0.01\,x, x)$

  Leaky ReLU modifikacija je ReLU aktivacijske funkcije koja rješava problem neaktivnog stanja neurona u mreži "propuštanjem" malih vrijednosti umjesto pretvaranja negativnih vrijedosti u 0 kao kod ReLU funkcije.



Slika 1.6: Graf Leaky ReLU funkcije

- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$
  Sigmoid je nelinearna, diferencijabilna funkcija koja za izlaz ima vrijednosti iz intervala $(0, 1)$ te se koristi pri rješavanju problema binarne klasifikacije. Jedan od problema ove funkcije je mogućnost pojave nestajućih gradijenata. Ako *sigmoid* za ulaznu vrijednost dobija jako velike ili jako male vrijednosti, izlazna vrijednost se približava 1 ili 0 te će u tom slučaju gradijenti bili vrlo mali tijekom povratne propagacije. Više o problemu nestajućih gradijenata govorit ćemo u podpoglavlju 2.1. Isto tako, zbog računanja eksponencijalne funkcije prilikom aktivacije, *sigmoid* povećava kompleksnost računalnih operacija u usporedbi s drugim aktivacijskim funkcijama. Iz ovih razloga, *sigmoid* često više nije u uporabi.

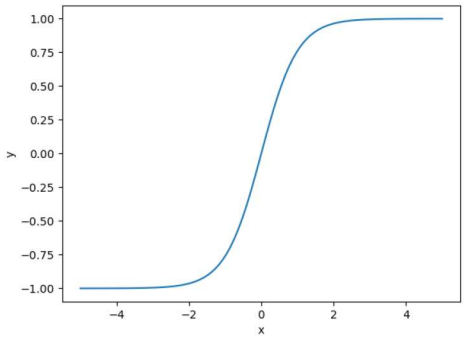


Slika 1.7: Graf sigmoid funkcije

- **Tangens hiperbolni**: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

  Tangens hiperbolni je nelinearna, diferencijabilna funkcija koja za izlaz ima vrijednost iz intervala $(-1, 1)$. Slična je oblikom i uporabom kao i sigmoid funkcija, ali zbog svoje centriranosti oko 0, pogodnija je za optimizaciju. Nedos-

tatci su slični kao i kod sigmoid funkcije, poput pojave nestajućih gradijenata i računske kompleksnosti.



Slika 1.8: Graf tanh funkcije

Osim navedenih aktivacijskih funkcija, važno je i spomenuti *softmax* funkciju koja nije aktivacijska funkcija jer se ne primjenjuje na izlaz pojedinog neurona nego na cijeli sloj te se često koristi na izlaznom sloju pri rješavanju višeklasnih klasifikacijskih problema.
Softmax funkcija definirana je s izrazom:

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, i = 1 \dots n$$

Softmax preslikava vektor realnih brojeva u vektor s vrijednostima iz segmeta $[0, 1]$ koji u sumi imaju vrijednost 1 tj. svaki element tog vektora će tada predstavljati vjerojatnost da ulaz pripada klasi koju taj element predstavlja.

1.2 Tipovi učenja i treniranje neuronskih mreža

U prethodnim potpoglavljima definirali smo *feed forward* mreže i pokazali kako podatci s ulaznog sloja dolaze do izlaznog. Ova procedura nam nije korisna ako ne znamo definirati učenje mreža i prilagoditi treniranje mreže za problem koji želimo rješiti.

Postoje tri osnovna tipa učenja:

- **Nadzirano (*eng. supervised*) učenje:**
  U ovom obliku učenja, neuronska mreža trenirana je na podatcima za koje znamo željeni izlaz (*eng. labeled data*). Treniranje mreže temelji se na predviđanju točnog izlaza za dane ulazne podatke uspoređujući izlaze mreže s točnim izlazima. Nadzirano učenje najčešći je oblik učenja i koristi se u širokom spektru problema kao što su primjerice klasifikacijski i regresijski problemi.

- **Nenadzirano (*eng. unsupervised*) učenje:**
  U ovom obliku učenja, za razliku od nadziranog učenja, mreža je trenirana na podatcima za koje nam nisu poznate prave vrijednosti izlaza. Nenadzirano

učenje često se koristi pri redukciji dimenzionalnosti podataka. Redukcija dimenzionalnosti podataka smanjuje broj značajki u podatcima, a pri tom zadržava bitne značajke. Jedna od najčešćih pristupa konstrukciji mreža koje vrše redukciju dimenzionalnosti je autoenkoder, koji se sastoji od dviju neuronskih mreža - enkodera i dekodera. U ovom pristupu, cilj mreže je rekonstruirati svoj ulaz što bolje. Enkoder uzima ulazne podatke mreže i preslikava ih u reprezentaciju podataka manjih dimenzija. Dekoder potom pokušava rekonstruirati tu reprezentaciju nazad u ulazne podatke. Funkcija gubitka će tada predstavljati razliku ulaznih i rekonstruiranih podataka te će cilj treniranja tada biti minimizirati tu razliku.

- **Učenje podrškom** (*eng. reinforcement learning*)**:**
  Ovaj oblik učenja temelji se na interakciji agenta s okolišem. Za odabranu akciju agenta u interakciji s okolišem, davaju se nagrade ili kazne. Neuronske mreže u ovom obliku učenja najčešće se koriste pri pronalaženju strategije za maksimiziranje dobivene nagrade. Mreža za ulazne podatke prima stanje okoliša, a za izlaz daje vjerojatnosnu distribuciju mogućih akcija agenta. Agent potom odabire akciju sukladno toj distribuciji i nagrada ili kazna se koristi kako bi se ažurirali parametri mreže. Učenje podrškom najčešće se koristi u igranju igara i u problemima za koje ne možemo analitički odrediti optimalno rješenje nego strategijom pokušaja i pogreške.

U nastavku, usredotočiti ćemo se na nadzirani oblik učenja jer je korišten pri izradi projektnog zadatka.

Kako bi uspješno trenirali mrežu u nadziranom tipu učenja, potrebno je definirati funkciju gubitka kako bi odredili ukupno odstupanje trenutnog izlaza mreže od danog točnog izlaza.
Neka je svaki ulazni podatak u mreži označen s $x \in \mathbb{R}^n$, a s $y \in \mathbb{R}^m$ označimo pripadajući točni izlaz mreže. Mreža će za dani ulaz x tada imati pripadnu točnu vrijednost $y(x)$ te dati izlaznu vrijednost:

$$\hat{y}(x; w)$$

Skup za treniranje tada možemo definirati kao skup uređenih parova $\{(x_i, y_i), i = (1, \dots N)\}$.
Funkcija

$$L(y(x), \hat{y}(x, w)) \tag{1.2}$$

predstavlja odstupanje izlaza mreže $\hat{y}(x, w)$ za danu točnu vrijednost y. Predikcija $\hat{y}(x, w)$ ovisi o ulaznim podatcima x i težinama w.

Funkcija gubitka (*eng. cost function*) vraća skalarnu vrijednost koja predstavlja prosjek odstupanja izlaza modela na cijelom skupu za treniranje i definirana je s izrazom:

$$J(w) = \frac{1}{N} \sum_{i=1}^{N} L(y_i(x), \hat{y}_i(x, w)) \tag{1.3}$$

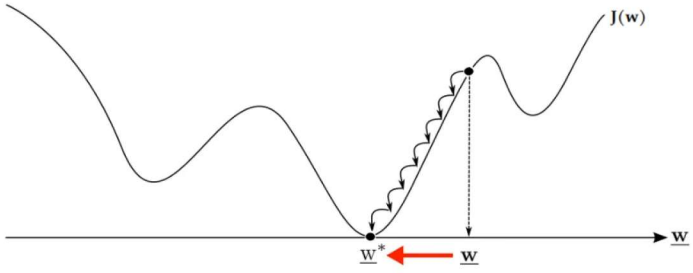
Tablica 1.1 daje nam pregled često korištenih funkcija gubitka u regresijskim i klasifikacijskim problemima. Osim navedenih, postoje mnoge druge te njihov odabir ovisi o zadanom problemu i modelu koji odlučimo koristiti.

Funkcija gubitka - regresija	**Formula**
Mean Squared Error (MSE)	$J(w) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
Root MSE	$J(w) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$
Mean Absolute Error (MAE)	$J(w) = \frac{1}{N}\sum_{i=1}^{N}\lvert y_i - \hat{y}_i\rvert$
Funkcija gubitka - klasifikacija	**Formula**
Cross Entropy (binarna klasifikacija)	$J(w) = -\sum_{i=1}^{N}(y_i \log(\hat{y}_i) + (1+y_i)\log(1-\hat{y}_i))$
Cross Entropy (klasifikacija - k klasa)	$L(y,\hat{y}) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$ $J(w) = \frac{1}{N}\sum_{i=1}^{N} L(y_i,\hat{y}_i)$

Tablica 1.1: Najčešće funkcije gubitka

1.2.1 Učenje gradijentnim spustom

Cilj treniranja mreže je minimizirati funkciju gubitka tako da pronađemo vrijednosti težina w za koje je razlika između predviđene i željene vrijednosti minimalna. Minimizacija se odrađuje korištenjem optimizacijskih algoritama poput gradijentnog spusta (*eng. gradient descent*).
Gradijentni spust je iterativna metoda u kojoj nam je cilj, počevši od neke točke, mijenjati vrijednosti težina tako da se vrijednost funkcije gubitka minimizira. Uzimajući u obzir graf funkcije gubitka, "spuštamo" se od neke točke prema nižim vrijednostima dok ne dođemo do minimuma.



Slika 1.9: Graf funkcije gubitka s minimumom u w^*

$$\begin{aligned} \{(x_i, y_i), i = 1 \dots N\} &\quad \rightarrow \text{skup za treniranje} \\ J(w) = \frac{1}{N} \sum_{i=1}^{N} L(y(x), \hat{y}(x, w)) &\quad \rightarrow \text{funkcija gubitka} \\ \mathbf{w}^* = \underset{w}{argmin}\, J(w) &\quad \rightarrow \text{minimum} \end{aligned} \tag{1.4}$$

Ako bi smo htjeli izračunati optimalne vrijednosti težina analitički, krenuli bismo od računanja gradijenta funkcije gubitka. U slučaju perceptron modela s jednim neuronom, za težine $w = (w_0, w_1, w_2 \dots w_n)^T$, gradijent funkcije gubitka računali bi uzeći prve derivacije komponenata po varijabli w.

$$\frac{\partial J(w)^T}{\partial w} = \left(\frac{\partial J(w)^T}{\partial w_0}, \frac{\partial J(w)^T}{\partial w_1}, \dots \frac{\partial J(w)^T}{\partial w_n} \right)^T \tag{1.5}$$

Potom bismo izjednačili gradijent s nulom i tražili stacionarne točke i odredili globalni minimum. U primjeru modela s jednim neuronom ovaj pristup je opravdan, ali u slučaju gdje koristimo nediferencijabilne aktivacijske funkcije poput ReLU-a i imamo kompleksniju strukturu mreže, u praksi se koriste iterativne metode poput gradijentnog spusta.

Gradijentni spust pruža nam iterativni postupak za pronalaženje ovog minimuma. Počevši od nasumično odabrane vrijednosti težina, krećemo se u negativnom smjeru gradijenta. Korak ove metode definiran je s:

$$w_j(t+1) = w_j(t) - \eta \frac{\partial J(w)^T}{\partial w_j}, \quad j = 0 \dots n \tag{1.6}$$

gdje je η stopa učenja. Preostaje nam još pogledati kako ćemo izračunati gradijent funkcije gubitka.

$$\begin{aligned} \frac{\partial J(w)^T}{\partial w_j} &= \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(y(x), \hat{y}(x, w))}{\partial w_j} \\ &= \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(y(x), \hat{y}(x, w))}{\partial \hat{y}(x, w)} \cdot \frac{\partial \hat{y}(x, w)}{\partial w_j} \end{aligned} \tag{1.7}$$

Prvi faktor produkta u 1.7 ovisit će o odabiru funkciju gubitka dok će drugi faktor ovisiti o tipu modelu s kojim raspolažemo.
Nedostatak korištenja metode gradijentnog spusta je što u slučaju da optimizacijski problem nije konveksan i ovisno o početnoj točki, možemo konvergirati u lokalni minimum, a ne globalni. Osim toga, ako raspolažemo velikom količinom podataka, računanje gradijenta u svakoj iteraciji je vremenski skupo. Iz tih razloga, u praksi se koriste optimiziranije verzije gradijentnog spusta:

- Stochastic gradient descent (SGD): metoda u kojoj se nasumično bira jedan podatak i računa gradijent

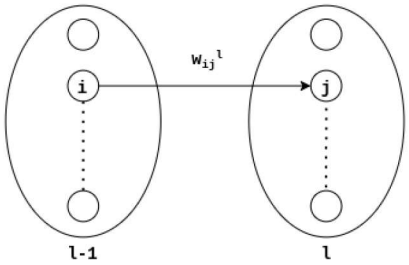
- Mini batch gradient descent: varijacija SGD metode u kojoj se skup podataka dijeli na manje skupove (*eng. mini batch*) te se računanje gradijenata i ažuriranje težina događa nakon obrade cijelog *mini batcha*

Osim različitih verzija algoritma gradijentnog spusta, koriste se i brojni optimizacijski algoritmi poput **AdaGrad**(*Adaptive Gradient Descent*) i **Adam**(*Adaptive Moment Estimation*). Pregled različitih optimizacijskih algoritama gradijentnog spusta može se pronaći u [4].

1.2.2 Algoritam povratne propagacije

Algoritam povratne propagacije koristi se za optimalnije računanje gradijenata funkcije gubitka s obzirom na težine unutar mreže. Algoritam nije vezan za specifičnu arhitekturu mreže te ćemo u nastavku pokazati općeniti način rada algoritma. Slojeve mreže označit ćemo se $l \in \{1 \dots L\}$. Označimo težinu koja povezuje $j-ti$ neuron u sloju l s neuronom i u sloju $(l-1)$ s w_{ij}^l. Gradijent koji želimo izračunati u višeslojnoj mreži tada poprima oblik:

$$\frac{\partial J(w)^T}{\partial w_{ij}^l} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial L(y(x), \hat{y}(x,w))}{\partial \hat{y}(x,w)} \cdot \frac{\partial \hat{y}(x,w)}{\partial w_{ij}^l} \tag{1.8}$$



Slika 1.10: Povezanost neurona j u sloju l s neuronom i u sloju l s težinom w_{ij}^l između njih.

Kako bi izračunali izraz $\frac{\partial \hat{y}(x,w)}{\partial w_{ij}^l}$, odredimo prvo ukupni ulaz koji dolazi u neuron j u sloju l. Označit ćemo ga s h_j^l i zovemo ga aktivnost neurona. Aktivnost ćemo računati kao težinsku sumu aktivacija neurona iz sloja $l-1$ koji su povezani s tim neuronom, aktiviranu s prikladnom funkcijom iz sloja l:

$$h_j^l = f^l(\underbrace{\sum_{i=1}^{n_{l-1}} w_{ij}^l h_i^{l-1} + b_j^l}_{z_j^l}) \tag{1.9}$$

gdje je:

- f^l - aktivacijska funkcija korištena u sloju l

- n_{l-1} - oznaka za broj neurona u prethodnom sloju $l-1$
- h_i^{l-1} - aktivnosti neurona u prethodnom sloju $l-1$
- b_j^l - oznaka za vrijednost bias neurona (njegove težine su sadržane u w)
- z_j^l - oznaka težinske sume bez aktivacije uvedena radi preglednosti u nastavku

Ako se vratimo na računanje drugog izraza u (1.8), sada imamo:

$$\frac{\partial \hat{y}(x,w)}{\partial w_{ij}^l} = \frac{\partial \hat{y}(x,w)}{\partial z_j^l} \cdot h_j^l \tag{1.10}$$

$$\delta_j^l := \frac{\partial \hat{y}(x,w)}{\partial z_j^l} \tag{1.11}$$

Izraz (1.11) predstavlja grešku neurona j u sloju l (eng. "local error") tj. to je pokazatelj utjecaja neurona j u sloju l na izlaz i ukupnu grešku mreže.
Grešku neurona u sloju l dalje možemo raspisati s:

$$\delta_j^l = \sum_{i=1}^{n_{l+1}} \frac{\partial \hat{y}(x,w)}{\partial z_i^{l+1}} \cdot \frac{\partial z_i^{l+1}}{\partial z_j^l} \tag{1.12}$$

Prvi izraz u sumi (1.12) će biti greška neurona u idućem sloju, tako da nam preostaje raspisati:

$$\frac{\partial z_i^{l+1}}{\partial z_j^l} = w_{ji}^{l+1}(f^{l+1})'(z_j^l) \tag{1.13}$$

Tada imamo:

$$\delta_j^l = \sum_{i=1}^{n_{l+1}} \delta_i^{l+1} w_{ji}^{l+1}(f^{l+1})'(z_j^l) \tag{1.14}$$

Sada kada smo definirali sve što nam je potrebno, pogledajmo dvije faze algoritma povratne propagacije:

- **Propagacija unaprijed** (*forward propagation*):
  U ovoj fazi računamo aktivnosti neurona počevši od ulaznog sloja prema izaznom.

$$h_j^0 := x_j \tag{1.15}$$

$$z_j^l = \sum_{i=1}^{n_{l-1}} w_{ij}^l h_i^{l-1} + b_j^l \tag{1.16}$$

$$h_j^l = f^l(z_j^l) \tag{1.17}$$

$$\hat{y}(x,w) = f^L(z^L) \tag{1.18}$$

  Aktivnosti funkcija u ulaznom sloju h_j^0 su vrijednosti ulaznih podataka (primjenjuje se aktivacijska funkcija identiteta) dok je aktivnost izlaznog sloja dana s $f^L(z^L)$, gdje je L oznaka izlaznog sloja.

- **Propagacija unazad** (*backpropagation*):
  U ovoj fazi računamo greške pojedinih neurona, počevši od izlaznog sloja prema početnom.

$$\delta_j^L = \frac{\partial \hat{y}(x,w)}{\partial z_j^L} = (f^L)'(z_j^L) \tag{1.19}$$

$$\delta_j^l = \sum_{i=1}^{n_{l+1}} \delta_i^{l+1} w_{ji}^{l+1} (f^{l+1})'(z_j^l) \tag{1.20}$$

  (1.19) predstavlja pogrešku u zadnjem sloju te gledajući aktivaciju izlaza (1.18) možemo vidjeti da će izraz kojime računamo tu pogrešku biti jednak vrijednosti derivacije aktivnosti neurona. U ostalim slojevima pogreške računamo kako je prethodno opisano izrazom 1.14.

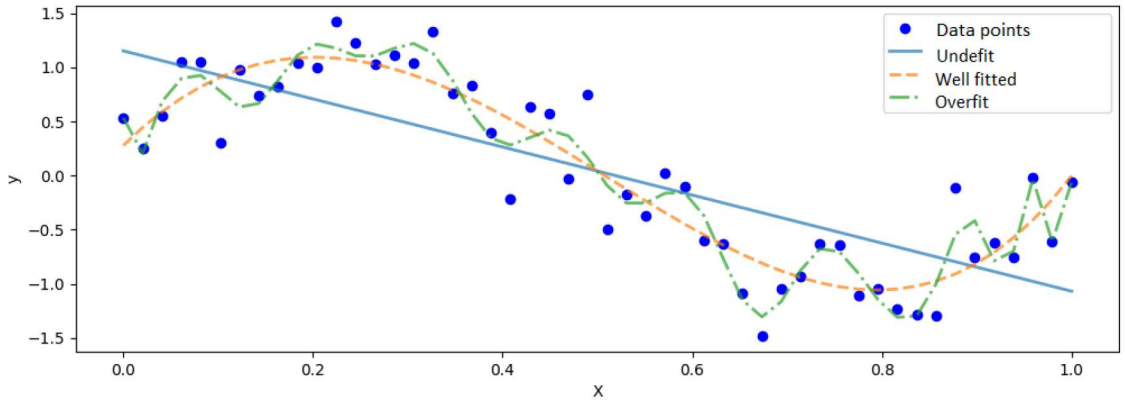
Sada kada smo izračunali aktivnosti neurona i njihove greške sve što preostaje je ažurirati vrijednosti težina u mreži:

$$w_{ij}^l \leftarrow w_{ij}^l - \eta \cdot \frac{\partial J(w)}{\partial w_{ij}^l} = w_{ij}^l - \eta \cdot \frac{\partial J(w)}{\partial \hat{y}(x,w)} \cdot \frac{\partial \hat{y}(x,w)}{\partial w_{ij}^l} \tag{1.21}$$

$$= w_{ij}^l - \eta \cdot \frac{\partial J(w)}{\partial \hat{y}(x,w)} \cdot \delta_j^l \cdot h_j^{l-1} \tag{1.22}$$

1.2.3 Problemi učenja mreža

Rezultati treniranja mreže ovise o mnogo parametara, a najčešći problemi koji se mogu pojaviti su prenaučenost (*overffiting*) i podnaučenost (*underffiting*). *Overffiting* je problem u kojemu se trenirani model previše prilagođava danom skupu za treniranje, a na novim podatcima ima znatno lošije performanse. *Overffiting* se događa kada je model presložen za problem koji rješava ili je preosjetljiv na šum u podatcima. *Underffiting* je suprotan problem od *overffitinga* - mreža nije dovoljno kompleksna za problem koji rješava, ima premalo parametara ili nije dovoljno dobro trenirana. Rezultat toga je da je problem presložen za model koji se upotrebljava i ostvaruje loše performanse i na skupu za treniranje i na novim podatcima.



Slika 1.11: Primjer regresije s tri razlilčita ishoda treniranja

Na slici 1.11 vidimo da u slučaju *underffit-a* (plava boja) aproksimacija mreže nedovoljno dobro opisuje pravilnost danih podataka, a u slučaju *overffit-a* (zelena boja) aproksimacija točno prati točke u danom skupu podataka, ali generalno ne aproksimira traženu funkciju ispravno. U *well fitted* slučaju (narančasta boja) imamo najbolju aproksimaciju funkcije koja opisuje dane podatke.

Kako bi se izbjegao *overfitting*, možemo trenirati na većem skupu podataka, smanjiti kompleksnost modela, dodati regularizaciju podataka i dr. Regularizacija je bilo kakva promjena u algoritmu učenja koja smanjuje grešku generalizacije, ali ne povećava grešku treniranja [2]. Za izbjegavanje *underfittinga* koristimo suprotne metode: nećemo povećavati skup podataka za treniranje nego kompleksnost modela, povećati broj parametara i slojeva modela.

Jedna od najčešćih metoda regularizacije za prevenciju *overfittinga* je *dropout* - metoda u kojoj je određeni postotak neurona tijekom treniranja nasumično izostavljen i neaktivan u toj iteraciji. Ovom tehnikom smanjujemo ovisnost neurona jedni od drugima te poboljšavamo generalizaciju modela jer je model natjeran naučiti uzorak u podatcima umjesto prilagođavanja i na taj način smanjuje vjerojatnost *overfittinga*.
Drugi problemi koji se mogu pojaviti tokom treniranja i dodatne metode za sprječavanje opisanih problema mogu se pronaći u poglavlju **1.4** u [1].

2 | Rekurentne neuronske mreže

Rekurentne neuronske mreže (*eng. Recurrent neural networks - RNN*) vrsta su neuronskih mreža koje se koriste za obradu sekvencijalnih podataka poput teksta, govora, bioloških podataka ili drugih vremenski ovisnih podataka. Za razliku od klasičnih neuronskih mreža koje imaju fiksnu veličinu ulaznih podataka, rekurentne mreže mogu raditi s varijabilnom veličinom ulaznih podataka. Glavna razlika rekurentnih u odnosu na ostale neuronske mreže je pojava petlji unutar arhitekture modela koja omogućuje zadržavanje informacija od prethodnih ulaznih podataka u mreži. Ovo svojstvo omogućuje nam rješavanje problema u kojima su podatci međusobno ovisni jedni o drugima za razliku od *feed forward* mreža u kojima je svaki ulazni podatak neovisan o drugome.

Primjer 2. *Pretpostavimo da želimo napraviti model koji će određivati sentiment filmskih recenzija. Za ulazne podatke koristimo bazu filmsku recenzija i želimo odrediti za danu filmsku recenziju je li recenzija pozitivna ili negativna.*

Kada bismo koristili *feed forward* neuronsku mrežu za rješavanje ovog problema za ulazne podatke mogli bismo uzeti vektorsku reprezentaciju svake pojedine riječi iz recenzije i za izlaz dobiti binarno predviđanje (negativno/pozitivno). Međutim, problem se pojavljuje kada uzmemo u obzir recenzije poput:

Film mi se sviđa, ali mi se ne sviđa kraj.

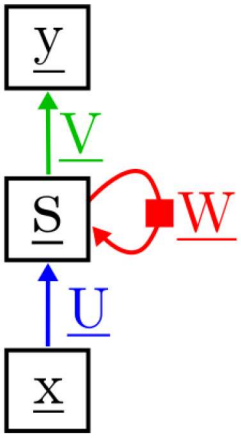
U ovoj recenziji *feed forward* mreža će prepoznati ključne riječi "sviđa" i "ne sviđa", ali ih neće moći staviti u kontekst rečenice tj. neće moći prepoznati da je općeniti dojam filma pozitivan, ali specifični dio filma poput kraja negativan. Rekurentna neuronska mreža može uzeti u obzir poziciju ovih riječi i s obzirom na njihovu okolinu odrediti kontekst u kojem se riječi pojavljuju.

Primjer 3. [2] *Uzmimo u obzir rečenice:*

Putovao sam u Nepal 2009. godine.
2009. godine, putovao sam u Nepal.

Obje rečenice imaju isto značenje, ali se razlikuju u strukturi rečenice tj. redoslijed im je obrnut. *Feed forward* mreža moći će prepoznati 2009. godinu kao relevantnu informaciju, ali se u prvom slučaju pojavljuje na kraju rečenice, a drugoj na početku. Ako koristimo mrežu koja za ulaz ima fiksnu duljinu rečenice, ovisno o toj duljini moguće je da nećemo pronaći informaciju o godini putovanja, a treniranje različitih mreža za različite duljine rečenica nije isplativo.

Sada kada smo pokazali na primjerima potrebu za korištenjem rekurentnih neuronskih mreža, pogledajmo osnovnu strukturu *RNN* mreže. Označimo s $x^{(t)} \in \mathbb{R}^n$ ulazni vektor u nekom trenutku t, s $U \in \mathbb{R}^{h \times n}$ matricu težina između ulaznog sloja i skrivenog sloja, gdje je h broj neurona u skrivenom sloju. Neka je $W \in \mathbb{R}^{h \times h}$ matrica težina koju skriveni sloj koristi prilikom preslikavanja u samog sebe i $V \in \mathbb{R}^{m \times h}$ matrica težina između skrivenog sloja i izlaznog, gdje je m broj izlaznih vrijednosti mreže.



Slika 2.1: Osnovna arhitektura RNN mreže

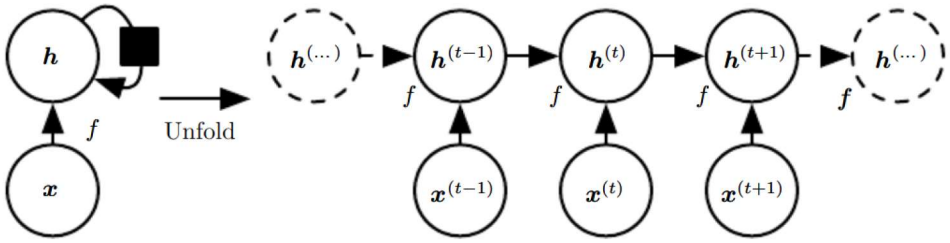
Na slici 2.1 možemo vidjeti već spomenutu pojavu petlji u strukturi rekurentnih mreža. Posljedica toga je da aktivnost neurona u skrivenom sloju S neće ovisiti samo o ulazu $x^{(t)}$ nego i o vrijednostima neurona u sloju S iz prethodnog koraka sekvence $t-1$. Potrebno je napomenuti da oznaka t ne mora nužno označavati vremensku jedinicu nego može biti i pozicija unutar sekvence podataka. Aktivacije neurona i propogacijski korak unaprijed tada ćemo računati s izrazima:

$$s_i^{(t)} = \tanh\left(\sum_{k=1}^{n} U_{ik}^T x_k^{(t)} + \sum_{j=1}^{h} W_{ij}^T s_j^{(t-1)} + b_i^s\right) \tag{2.1}$$

$$y_k^{(t)} = f\left(\sum_{j=1}^{h} V_{kj}^T s_j^{(t)} + b_k^y\right) \tag{2.2}$$

S obzirom da se RNN mreže najčešće koriste u obradi teksta, u izrazu (2.1) podrazumijevamo aktivacijsku funkciju tangens hiperbolni za skrivene slojeve. Aktivacijska funkcija za izlazni sloj može primjerice biti *softmax* funkcija ako se radi o klasifikaciji.

Stukturu rekurentnih mreža možemo promatrati i u obliku "odmotanog" grafa u kojem svaki vremenski korak t promatramo kao samostalni čvor grafa. Ovim načinom jasnije možemo vidjeti ovisnost izlaza skrivenih slojeva povezanih rekurzijom o slojevima u prethodnom koraku.



Slika 2.2: Primjer "odmotanog" grafa dijela RNN mreže u kojoj se događa rekurzija.

Iz osnovne strukture možemo konstruirati različite vrste RNN mreža. Neki primjeri dizajna RNN mreža su:

- RNN mreža koja u svakom koraku t izbacuje izlazni podatak i pojava rekurentnosti se događa između skrivenih slojeva.
- RNN mreža koja u svakom koraku t izbacuje izlazni podatak i pojava rekurentnosti se događa između izlaza mreže u trenutku t i skrivenih slojeva u idućem koraku $t + 1$.
- RNN mreža u kojoj se rekurentnost pojavljuje između skrivenih slojeva mreže i koja nakon obrade sekvence izbacuje jedan izlaz za cijelu sekvencu.

2.1 Treniranje RNN mreža

U izrazima (2.1) i (2.2) pokazali smo kako računati aktivnosti neurona u RNN mreži. Kao i u slučaju s *feed forward* mrežama, pokazati ćemo kako koristiti aktivnosti neurona i modificirati algoritam povratne propagacije kako bi uspješno trenirali rekurentne mreže. Ako bi pokušali primjeniti algoritam povratne propagacije na RNN mrežu, problem koji bi uočili je ponavljanje težina između slojeva mreže. Kada smo definirali povratnu propagaciju za *feed forward* mreže, pretpostavka je bila da su težine između slojeva različite i slojevi ne ovise jedan o drugome. Rješenje ovog problema je promatrati RNN mrežu u obliku "odmotanog" grafa kao što je prikazano u (2.2). U ovom obliku, RNN mrežu možemo gledati kao *feed forward* mrežu u kojoj će svaki korak kroz vijeme predstavljati jedan "sloj" mreže i tada možemo pretpostaviti da su sve težine $W^{(t)}$, $U^{(t)}$ i $V^{(t)}$ u koraku t neovisne jer su kopije istih težinskih matrica, ali u drukčijem vremenskom koraku. Gradijente tada možemo zbrojiti nakon propagacije unatrag. Algoritam koji je nastao iz opisane ideje zove se **povratna propagacija kroz vrijeme** (*eng. back-propagation through time - BPTT*) i koristi se za treniranje rekurentnih mreža.

Koraci ovog algoritma su:

- Računanje aktivnosti neurona u mreži i računanje grešaka za svaki vremenski korak t

- Računanje gradijenata u propagaciji unazad ne uzimajući u obzir da su težine dijeljene između vremenskih koraka. Za $t = 1, \dots T$ prepostavljamo da su težine $W^{(t)}$, $U^{(t)}$ i $V^{(t)}$ neovisne i računamo gradijente $\frac{\partial J}{\partial U^{(t)}}$, $\frac{\partial J}{\partial W^{(t)}}$ i $\frac{\partial J}{\partial V^{(t)}}$ koristeći standardni algoritam povratne propagacije.
- Zbrajanje svih gradijenata dijeljenih težina u odnosu na vremenske korake:

$$\frac{\partial J}{\partial U} = \sum_{t=1}^{T} \frac{\partial J}{\partial U^{(t)}}$$
$$\frac{\partial J}{\partial W} = \sum_{t=1}^{T} \frac{\partial J}{\partial W^{(t)}}$$
$$\frac{\partial J}{\partial V} = \sum_{t=1}^{T} \frac{\partial J}{\partial V^{(t)}}$$

Nedostatak korištenja povratne propagacije kroz vrijeme je što u slučaju dugih sekvenci, algoritam može imati visoku vremensku složenost. S obzirom da promatramo mrežu u obliku "odmotanog" grafa, duljina sekvence će predstavljati broj skrivenih slojeva. Vremenska složenost je tada proporcionalna broju slojeva (duljini sekvence) pomnoženim s brojem neurona u skrivenom sloju. Za duge sekvence, računanje svih gradijenata tada postaje računski skupo i neefikasno. Jedan od prijedloga za rješavanje ovog problema možemo pronaći u [1] u obliku *krnjeg* algoritma povratne propagacije kroz vrijeme u kojem se umjesto računanja svih gradijenata tijekom povratne propagacije, gradijent računa samo na segmentu sekvence. S obzirom da samo gubitak na odabranom relevantnom segmentu sekvence pridonosi treniranju mreže, smanjujemo vremensku složenost, ali zato dobijamo aproksimaciju gradijenta.

Osim vremenske složenosti BPTT algoritma, treniranje rekurentnih neuronskih mreža ima i druge probleme, prvenstveno **problem nestajućeg/eksplodirajućeg gradijenta**. Jednostavne RNN mreže više uče iz pogrešaka koje su blizu trenutnog koraka sekvence nego iz udaljenijih pogrešaka pri početku sekvence. Primjerice, pri obradi dugačkog teksta, ako je relevantna informacija koju želimo uhvatiti pri početku teksta, utjecaj gradijenta s početka sekvence će biti beznačajan u usporedbi s kasnijih gradijentima i time će onemogućiti pravilno treniranje mreže.
Uzmimo u obzir jednostavnu RNN mrežu (kao što je prikazana na (2.1)) s linearnom aktivacijskom funkcijom između skrivenih slojeva i pogledajmo kako se aktivnosti skrivenih neurona mijenaju tokom ponavljanja istog rekurentnog koraka. Ulaz i izlaz mreže možemo zanemariti, pretpostavimo samo da smo dobili jednu ulaznu vrijednost i da ponavljamo rekurentni korak. Aktivnost skrivenog sloja da će biti:

$$s^{(t)} = Ws^{(t-1)} = (W)^t s^{(0)}$$

Potom, napravimo dekompoziciju matrice $W \in \mathbb{R}^{h \times h}$:

$$W = Q \Lambda Q^T$$

gdje je $Q \in \mathbb{R}^{h \times h}$ ortogonalna matrica, a $\Lambda \in \mathbb{R}^{h \times h}$ matrica koja na svojoj dijagonali ima svojstvene vrijednosti matrice W. Tada imamo:

$$s^{(t)} = Q(\Lambda)^t Q^T s^{(0)}$$

Aktivnosti neurona skrivenog sloja će s dovoljnim brojem koraka sekvence biti dominirane najvećom svojstvenom vrijednosti $|\lambda_1|$.

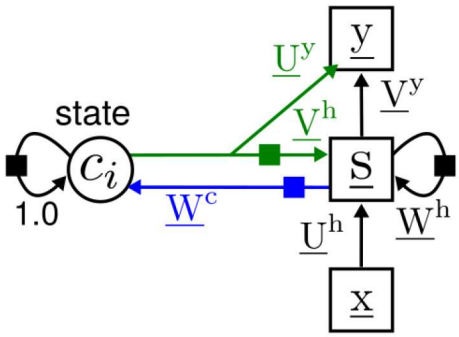
$$|\lambda_1| > 1 \rightarrow \lim_{t \rightarrow \infty} s_i^{(t)} = \pm\infty, \forall i \qquad (\textit{eksplodiranje aktivnosti})$$

$$|\lambda_1| < 1 \rightarrow \lim_{t \rightarrow \infty} s_i^{(t)} = 0, \forall i \qquad (\textit{nestajanje aktivnosti})$$

Za dovoljnu duljinu sekvence, ako je najveća svojstvena vrijednost matrice težina skrivenog sloja veća od 1, aktivnosti neurona u sloju će težiti prema $\pm\infty$ i postat će nerelevantni za treniranje mreže. U drugom slučaju, gdje je najveća svojstvena vrijednost manja od 1, aktivnosti neurona će poprimiti jako male vrijednosti i težiti prema 0 i utjecaj na treniranje mreže će biti beznačajan.
Jedno od rješenja ovog problema za treniranje RNN mreža dano je u obliku **LSTM** (*eng. long short term memory*) ćelija kojima ćemo se baviti u idućem poglavlju.

2.2 LSTM

LSTM arhitektura nadogradnja je osnovne arhitekture rekurentnih mreža s ciljem rješavanja problema nestajućih/eksplodirajućih gradijenata. Osnovna ideja ove nadogradnje je uvođenje ćelija (*eng.memory state cells*) $c^{(t)}$ u strukturu mreže koje imaju mogućnost selektivnog zadržavanja relevantnih informacija iz prethodnih vremenskih iteracija mreže. Ove ćelije mogu utjecati na izlaz mreže i na aktivaciju skrivenog sloja u trenutnom vremenskom koraku t. Svaka ćelija mijenja svoju vrijednost ovisno o vrijednosti ćelije i skrivenom sloju iz prethodnog vremenskog koraka.



Slika 2.3: Uvođenje LSTM ćelija u arhitekturu

Matrice W^c, U^y, V^h na slici 2.3 predstavljaju težinske matrice koje povezuju ćeliju s ostatkom mreže. Na ćeliju utječe i parametar kojime određujemo koliko prethodne vrijednosti uzimamo od ćelije iz prethodnog sloja, u ovom slučaju uzeli smo

vrijednost 1.0 tj. uzeli smo u obzir cijelu prethodnu ćeliju. Kako bi od ove arhitekture došli do pravog LSTM-a potrebno je uvesti mehanizme selektivnog ažuriranja naše memorije u ćelijama u obliku ulaznih, izlaznih sklopova i sklopova zaboravljanja.

- Ulazni sklop (*eng. input gate*) $g^{i(t)}$: odlučuje koliko novog ulaza ulazi u memoriju ćelije

- Izlazni sklop (*eng. output gate*) $g^{o(t)}$: odlučuje koliko memorije ćelije prolazi u skrivene slojeve i izlaz mreže

- Sklop zaboravljanja (*eng. forget gate*) $g^{f(t)}$: odlučuje koliko memorije prethodnog stanja ćelije se propušta u trenutno stanje

Uvođenjem ovih sklopova možemo pogledati niz formula koje opisuju potpunu arhitekturu LSTM-a:

$$\boldsymbol{c^{(t)}} = g^{f(t)} \odot c^{(t-1)} + g^{i(t)} \odot \tanh(W^c s^{(t-1)}) \tag{2.3}$$

$$\boldsymbol{y^{(t)}} = f(V^y s^{(t)} + b^y + U^y(c^{(t)} \odot g^{o(t)})) \tag{2.4}$$

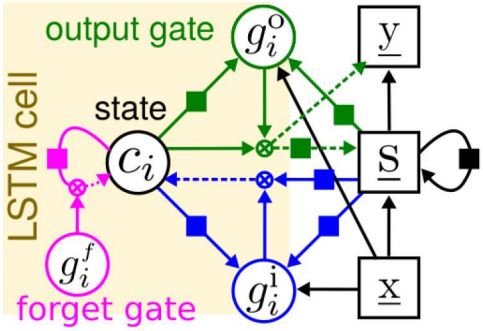
$$\boldsymbol{s^{(t)}} = \tanh(U^h x^{(t)} + W^h s^{(t-1)} + b^s + V^h(c^{(t-1)} \odot g^{o(t)}) \tag{2.5}$$

$$\boldsymbol{g^{i(t)}} = \sigma(U^i x^{(t)} + V^i c^{(t-1)} + W^i s^{(t-1)} + b^i) \tag{2.6}$$

$$\boldsymbol{g^{o(t)}} = \sigma(U^o x^{(t)} + V^o c^{(t-1)} + W^o s^{(t-1)} + b^0) \tag{2.7}$$

$$\boldsymbol{g^{f(t)}} = \sigma(U^f x^{(t)} + V^f c^{(t-1)} + W^f s^{(t-1)} + b^f) \tag{2.8}$$

Plavom bojom označili smo dio koji se odnosi na zapisivanje trenutne ćelije dugoročne memorije, zelenom bojom dijelove koji se odnose na utjecaj memorije na trenutni skriveni sloj i izlaz mreže, a ružičastom mehanizam zaboravljanja koji utječe na količinu informacija koje izostavljamo iz dugoročne memorije u trenutnoj ćeliji u odnosu na prošlu. Umjesto skalarne vrijednosti koja određuje koliko od prošlog stanja dugoročne memorije zadržavamo, uvodimo sklopku za zaboravljanje. Sklopove definiramo kao linearnu kombinaciju ulaznih stanja, utjecaja ćelije iz prošlog vremenskog koraka, skrivenog stanja iz prošlog vremenskog koraka i *bias parametara*. Za aktivaciju sklopki ima smisla koristiti *sigmoid* funkciju jer ona preslikava vrijednosti u interval $[0, 1]$, s čime dobijemo svojevrsni postotak koliko informacija zadržavamo, propuštamo dalje i zaboravljamo.



Slika 2.4: Grafički prikaz LSTM ćelije

Jedan od nedostataka LSTM arhitekture je što uvođenjem memorijskih ćelija i novih težinskih matrica povećavamo kompleksnost modela čime se povećava vjerojatnost da model *overfitt-a*. LSTM arhitekture zato često zahtjevaju velike količine podataka za treniranje kako bi se smanjio utjecaj kompleknosti na *overffiting*. Postoje i varijacije na arhitekturu LSTM-a koji se bave rješavanjem problema nestajućih/eksplodirajućih gradijenata poput **GRU** (*eng. gated recurrent unit*). GRU je pojednostavljena verzija LSTM-a sa sklopovnim mehaniznom. Više o GRU možete pronaći u [1], poglavlje 7.6.

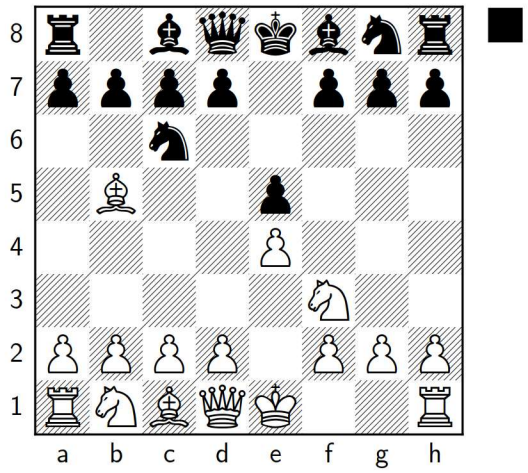
3 | Predviđanje ishoda šahovske igre

U ovom poglavlju bavit ćemo se projektnim zadatkom ovog rada tj. konstrukcijom neuronskog modela temeljenog na rekurentnim neuronskim mrežama koji će za dani niz ulaznih podataka poteza šahovske igre pokušati odrediti ishod igre. Cilj modela je odgovoriti na pitanje može li se šahovska igra obraditi kao niz podataka koristeći neuronske mreže i utječu li različite reprezentacije šahovske ploče na rezultate. Prije definiranja samog modela, potrebno je odrediti na koji način ćemo reprezentirati naše podatke u memoriji.

3.1 Reprezentacija šahovske ploče

Algebarska šahovska notacija najkorištenija je notacija danas i temeljena je na zapisivanju poteza na način da se prvo zapisuje figura s kojom je odigran potez te potom polje na koje je ta figura stavljena. Primjerice, ako bi smo htjeli prikazati poznato šahovsko otvaranje *Ruy Lopez* ili španjolska igra, zapisali bi ga ovako:

1 e4 e5 2 ♘f3 ♘c6 3 ♗b5

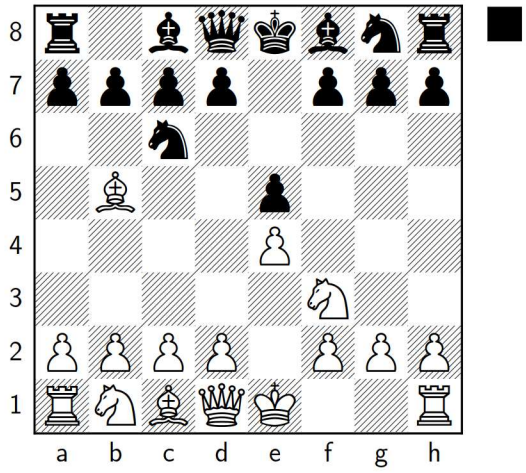


Šahovska polja označena su horizontalno slovima a-h i vertikalno brojevima 1-8. Svaki potez anotira se rednim brojem i počinje s potezom igrača s bijelim figurama te potom igrača s crnim figurama. Svaka figura označena je simbolom osim pijuna: kralj - K (*eng. king*), kraljica - Q (*eng. queen*), lovac - B (*eng. bishop*), skakač - N (*eng. knight*) i top - R (*eng.rook*). U danom primjeru *Ruy Lopez-a*, igra započinje pomicanjem *e* pijuna na polje *e4* na što igrač s crnim figurama odgovara

s pomicanjem pijuna na *e4* itd. Otvaranje bi tada bilo niz poteza zapisano s: **1.e4 e5 2.Nf3 Nc6 3.Bb5**. S obzirom da svaki redni broj sadrži poteze oba igrača, iz ove notacije možemo i zaključiti da je u našem primjeru igrač s crnim figurama na potezu. Dodatne oznake u ovoj notaciji uključuju i specificiranje figure ako primjerice imamo dva topa koja mogu doći na isto polje (npr. Rdf8 - Rd označava da se top na stupcu d pomiče), znak **x** ako se radi o uzimanju figure (npr. Bxe4 - lovac uzima figuru na polju e4), **O-O** i **O-O-O** za malu i veliku rošadu, znak **+** za šah i na kraju igre **1-0** ako je bijeli pobijedio, **0-1** ako je crni i **1/2 - 1/2** ako je remi. Gotovo sve *online* baze podataka sa šahovskim igrama koriste PGN (*eng. Portable Game Notation*), format u kojem dobijemo metapodatke igre poput datuma, turnira, imena igrača i ELO rating te samu igru u spomenutoj algebarskoj notaciji. Sada kada smo definirali oblik u kojem ćemo pronaći šahovske igre, potrebno je definirati prikladne reprezentacije stanja ploče za treniranje neuronskih mreža.

Fokusirat ćemo se na dva oblika reprezentacije šahovske ploče:

- **Bitmap** reprezentacija: u ovom obliku, šahovska ploča prikazana je binarnim vektorom duljine 12. Svaki element vektora prikazuje jednu figuru na ploči za bijelog i crnog igrača što odgovara broju 12 (6 figura po igraču: pijun, kralj, kraljica, top, skakač, lovac). Ukupna veličina vektora koji opisuje cijelo stanje ploče tada je $12 \times 64 = 768$. Na grafičkom prikazu ispod možemo vidjeti primjer bitmap reprezentacije na već prikazanom španjolskom otvaranju.



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdots \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdots$$

Bijeli pijuni Crni skakači

Matrica za svaku figuru je zapravo samo pokazatelj na kojim poljima na ploči se pojavljuju figure i ne prenosi druge informacije o ploči. Ipak, ova reprezentacija je često korištena jer je vrlo memorijski jeftina i brza te se pokazuje efektivna u neuronskim modelima.

- **Algebarska** reprezentacija: nadogradnja na *bitmap*, ali osim što prikazuje poziciju figura na ploči, isto tako daje svakoj figuri numeričku vrijednost. Tako primjerice pijuni vrijede 1, lovci i skakači 3, topovi 5, kraljica 9 i kralj 10. Ovaj prikaz pozicije prenosi više informacija o ploči jer osim razmještaja figura govori i njihovoj vrijednosti tj. jačini (npr. pijun vrijedi manje od lovca). Isto kao i kod *bitmap-a*, ukupna duljina ulaznih podataka jedne pozicije će biti 768. Pri treniranju neronskog modela, provjerit ćemo ima li ova nadogradnja pozitivan utjecaj na rezultate mreže u odnosu na *bitmap*. Kada bi prikazali isto otvaranje kao i u primjeru s *bitmap* reprezentacijom, dobili bi sljedeće:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdots \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \end{bmatrix} \cdots$$

Bijeli pijuni Bijeli topovi

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdots \begin{bmatrix} 0 & 0 & 0 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

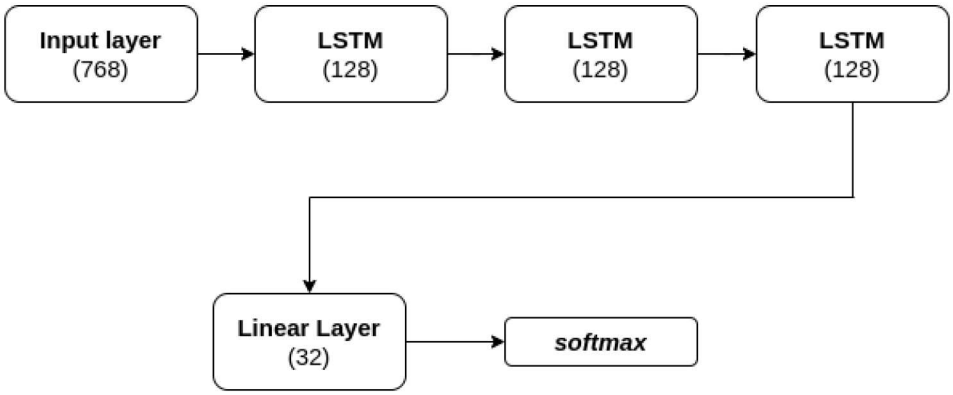
Crni skakači Crni kralj

Sada kad smo definirali načine na koje ćemo prikazati šahovske pozicije u memoriji, potrebno je opisati skup podataka za treniranje i definirati arhitekturu modela koja će obrađivati sekvence prikazane u tim oblicima. Podatke ćemo dobiti u već spomenutom *PGN* formatu tj. samu šahovsku igru u algebarskoj notaciji kao niz poteza. U pripremi podataka za treniranje svaka pozicija na ploči koja proizlazi iz niza poteza će biti pretvorena u *bitmap* ili *algebarske* tenzore koji će biti ulazni podatci mreže.

3.2 Metode i arhitektura modela

Stuktura modela u projektnom zadatku napravljena je po uzoru na [5]. Model je implementiran korištenjem *PyTorch* [7] biblioteke za neuronske mreže u Python programskom jeziku. Za model je odabrana već opisana LSTM arhitektura jer je pogodna za učenje sekvencijalnih podataka. Model se sastoji od:

- Ulaz mreže: *bitmap* ili *algebarski* tenzori duljine 768 (12x8x8)
- LSTM slojevi: 3 sloja s 128 ćelija s tangens hiperbolni aktivacijskom funkcijom
- Linearni sloj: sloj s 32 ćelije s ReLU aktivacijskom funkcijom
- Klasifikacijski sloj: sloj s 3 ćelije koje odgovaraju trima mogućim ishodima igre (pobjeda bijelog, crnog i remi) s *softmax* aktivacijom



Slika 3.1: Vizualni prikaz modela

Za optimizaciju korišten je **Adam** optimizator s adaptivnom stopom učenja, počevši s vrijednošću 0.1. Adaptivna stopa učenja je implementirana zbog sklonosti *overfittingu* LSTM mreža. Osim adaptivne stope učenja, korišteni su iz istog razloga i ***dropout*** slojevi između LSTM slojeva s vjerojatnošću ispuštanja neurona 0.2. Za funkciju gubitka, odabrana je *Cross-Entropy* funkcija gubitka u obliku za klasifikaciju k klasa (u slučaju ovog modela klasificiramo u 3 klase) koja je već prikazana u tablici 1.1.

Model je treniran na bazi podataka preuzetih s online baze podataka **FICS Games Database** [10]. Podatci su filtrirani tako da sadržavaju igre igrane u standardnom šahovskom vremenu ("duži format igre"). Ovaj vremenski format je odabran kako bi podatci bili vjerodostojniji s obzirom da u kraćim vremenskim formatima kvaliteta igara opada s zbog vremenskog pritiska na igrače te se češće događaju pogreške. Isto tako, podatci sadržavaju igre igrača čiji je prosječni **ELO** stupanj iznad 2000. ELO je sustav mjerenja jačine šahovskih igrača nazvan po svom izumitelju Arpadu Elo te je korišten u modernom šahu za rangiranje igrača. Za svaku odigranu igru šaha igrači gube ili dobijaju ELO bodove s obzirom na njihovu jačinu i jačinu njihovih protivnika. Više o ELO sustavu možete pročitati u [6]. ELO raspon

ljudskih igrača proteže se od 0 do 3000, a razina odabranog raspona iznad 2000 odabrana je zbog visoke kvalitete igara. Da smo odabrali igrače s manjim ELO-om (npr. u rasponu 500-1500), bilo bi teško učiti mrežu na takvim podatcima zbog oscilacija u kvaliteti i nepredvidljivosti ishoda. Primjerice, možemo imati igru u kojemu jedan igrač uvjerljivo pobjeđuje, ali na samom kraju igre napravi kardinalnu pogrešku i izgubi igru. Kada bismo predviđali ishod takve igre logično bi bilo predvidjeti da će igrač koji ima prednost pobijediti, ali bi u ovom slučaju pogriješili. Takvi slučaji manje su vjerojatni da će se dogoditi kod igrača s visokim ELO-om.
Fics Games Database podatci prikazani su u *PGN* formatu u kojem osim igre možemo pronaći i imena igrača, njihov ELO, datum odigrane igre, ime događaja na kojem se igra (u našem slučaju su to online igre) i koliko poteza u igri je napravljeni sa svakom figurom. Za naš model, jedino potrebno je sama igra, ali navedeni metapodatci mogu se iskorititi za potencijalne nadogradnje modela.

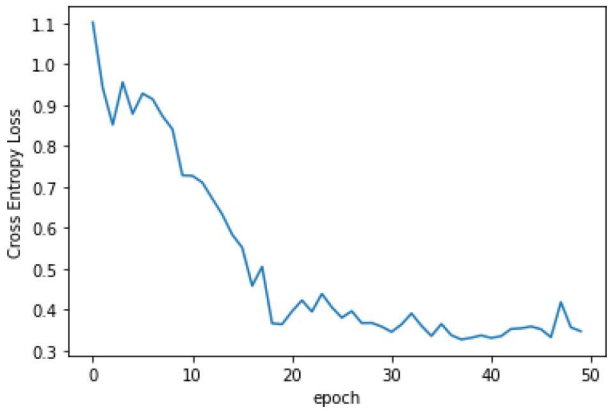
3.3 Treniranje modela i rezultati

Model je treniran i testiran na duljini igara 40 u dva različita slučaja: *bitmap* reprezentacijom i *algebarskom* reprezentacijom. Za treniranje i testiranje modela korišteno je:

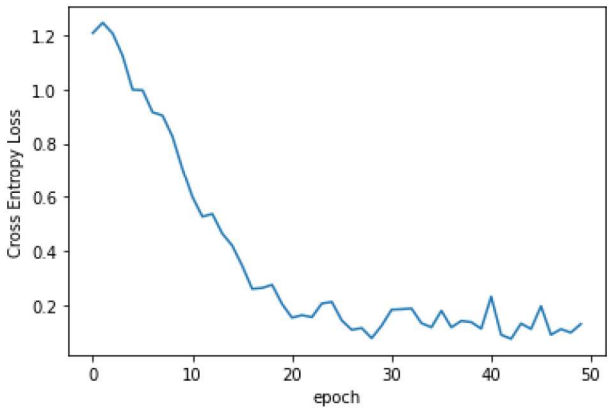
- Podatci za treniranje: 20 000 igara koje sadrže 9136 pobjeda bijelog igrača, 7901 pobjeda crnog igrača i 2963 remi-a
- Podatci za testiranje: 10 000 igara koje sadrže 4615 pobjeda bijelog igrača, 4001 pobjeda crnog igrača i 1384 remi-a

Podatci su odabrani da oba skupa imaju podjednako pobjeda bijelog i crnog igrača te da je $2/3$ podataka korišteno za treniranje, a $1/3$ za testiranje. Još jedan faktor koji treba uzeti u obzir je duljina sekvece tj. varirajuću duljinu igara s obzirom na broj poteza. Većina igara nema isti broj poteza, neke primjerice završavaju nakon 20-tak poteza ako jedan od igrača napravi kardinalnu porešku, a druge mogu biti i preko 80 poteza duge. S obzirom da će naš model primati fiksnu duljinu sekvenci, odabrali smo da će duljina igre biti 40 poteza, uzevši to kao prosječnu duljinu igre. Ako igra primjerice nema dovoljno poteza, nadopuniti ćemo igru s praznim tenziroma, a ako ima više poteza onda nećemo uzimati u obzir poteze nakon zadane granice.
U nastavku ćemo pogledati razlike u treniranju modela na već spomenutim reprezentacijama i vidjeti njihovu točnost na istom skupu testnih podataka.

Rezultati treniranja i usporedba funkcija gubitka tijekom epoha:



Slika 3.2: Gubitak modela tijekom treniranja na bitmap reprezentaciji



Slika 3.3: Gubitak modela tijekom treniranja na algebarskoj reprezentaciji

Iz grafova 3.2 i 3.3 možemo vidjeti da gubitak tijekom treniranja dviju reprezentacija je vrlo sličan te nema zamjetne razlike koja bi sugestirala da je jedna reprezentacija teža za trenirati od druge.
Iduće, pogledat ćemo točnost oba modela na testnom skupu podataka:

Reprezentacija	**Rezultat**	**Postotak**
Bitmap	5401/10000	54.01%
Algebarska	5351/10000	53.51%

Tablica 3.1: Točnost modela

Iz tablice 3.1 možemo primjetiti da je točnost modela s obje reprezentacije gotovo jednaka, ali *bitmap* reprezentacija usprkos tome što prenosi manje informacija o stanju ploče ostvaruje nešto bolje rezultate u odnosu na *algebarsku* iako za malu marginu. Autori [5] dobijaju slične rezultate - različiti eksprimenti dolaze do zaključka da su dvije reprezentacije vrlo slične u ovakvom tipu modela, ali u njihovom slučaju većinom *bitmap* reprezentacija ostvaruje bolje rezultate. Važno je napomenuti da iako smo s oba modela dobili rezultate oko 50% točnosti, ne radi se o

nasumičnom izboru rezultata jer s obzirom na tri moguća ishoda igre, nasumično biranje ishoda igre bi rezultiralo s 33.3% točnosti. Ovim rezultatima potvrđujemo pitanje postavljeno u uvodu projektnog rada - šah se može gledati i trenirati kao niz poteza te rezultati koje dobijemo su bolji od nasumičnog pogađanja. Isto tako bitno je uzeti u kontekst i sam zadatak modela. Predviđanje ishoda šahovske igre vrlo je komplesan problem s velikim brojem faktora koji utječu na ishod igre te uzimajući u obzir samo poteze u igri, postići vrlo visok postotak točnosti je gotovo nemoguć. Međutim, potvrda rada ovakvog modela predstavlja potencijal u obradi šahovskih igara na drukčiji način. Model se može nadograditi uzimajući u obzir metapodatke igara: imena igrača, njihov ELO rating, mjesta natjecanja, itd. Isto tako, treniranje na većem skupu podataka i povećanje kompleknosti mreže potencijalno bi doprinijelo radu modela. Napredak ovih nadogradnja potvrđeno je i u [5], gdje su autori dobili bolje rezultate koristeći metapodatke igara. U obzir se mogu uzeti i različite duljine igara na kojima se model trenira te kombinirajući rezultate na različitim duljinama možemo dobiti još objektivnije predviđanje. Model bi se mogao nadograditi s još metapodataka poput prijašnjih rezultata u međusobnim okršajima igrača, prijašnjim rezultatima igrača na istom turniru i vremenom potrošenim na svakom potezu igre što bi potencijalno moglo nadalje poboljšati rezultate modela.

Implementacija modela napisanog u Python programskom jeziku koristeći *PyTorch* [7] biblioteku nalazi se na Git repozitoriju [11].

4 | Zaključak

Šahovski modeli temeljeni na neuronskim mrežama imaju veliku ulogu u razvoju šahovske teorije i razumijevanju igre. Većina današnjih neuronskih šahovskih modela uzima u obzir poziciju na šahovskoj ploči i daju evaluaciju dane pozicije na temelju faktora poput materijalnog suficita/deficita igrača, razmještaja figura i pozicijske prednosti. Ta evaluacija omogućuje odabir najboljeg poteza u poziciji i optimalnom igranju šaha. U ovome radu, predložili smo alternativni pristup obrađivanju šahovske igre u kontekstu neuronskih mreža - sekvencijalnu obradu poteza igrača u svrhu predviđanja ishoda igre. U usporedbi s pristupom većine današnjih modela, gledanje šahovske igre kao sekvence poteza pri modeliranju neuronskih mreža je neistraženo područje. Za arhitekturu modela, odabrali smo LSTM arhitekturu rekurentnih neuronskih mreža koja je pogodna za obrađivanje sekvencijalnih podataka. Konstruirani model uspoređivali smo s obzirom na dvije predložene reprezentacije šahovske ploče u memoriji - *bitmap* i *algebarsku* reprezentaciju.
Rezultatima dobivenim u predloženom modelu potvrđujemo da se šahovska igra može obraditi kao niz poteza te da je dobivena točnost predviđanja ishoda igara veća od nasumičnog odabira. Na 10000 testnih podataka, za *bitmap* reprezentaciju ostvarili smo 54.01% točnosti, a za *algebarsku* 53.51%. *Bitmap* reprezentacija ostvaruje veću točnost na testnom skupu, što je iznenađujući rezultat s obzirom da *algebarska* reprezentacija prenosi više informacija o stanju ploče. Iako se dobiveni postotak točnosti ne čini visok, uzimajući u obzir da smo koristili samo poteze igrača, rezultati potvrđuju ispravnost koncepta modela. Predloženi model ima i prostora za napredak zbog svoje fleksibilnosti po pitanju reprezentacija ploče i mogućih nadogradnja obradom metapodataka igara.

Literatura

[1] C. C. Aggarwal, *Neural Networks and Deep Learning*, Springer, Yorktown Heights, NY, USA, 2018.

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

[3] N. Buduma, N. Lacascio, *Fundamentals of Deep Learning*, O'Reilly Media, USA, 2017.

[4] S. H. Haji, A. M. Abdulazeez, *COMPARISON OF OPTIMI-ZATION TECHNIQUES BASED ON GRADIENT DESCENT ALGORITHM: A REVIEW*, Duhok Polytechnic University, Duhok, Kurdistan Region, Iraq (2021).

[5] R. Dre˙zewski, G. W¸ator, *Chess as Sequential Data in a Chess Match Outcome Prediction Using Deep Learning with Various Chessboard Representati-ons*, AGH University of Science and Technology, Institute of Computer Science, Cracow, Poland (2021).

[6] M. E. Glickman, *A Comprehensive Guide To Chess Ratings*, Department of Mathematics, Boston University, USA (1995).

[7] Pytorch službena dokumentacija: `https://pytorch.org/docs/stable/`

[8] Dokumentacija biblioteke python-chess: `https://python-chess.readthedocs.io/`

[9] `https://www.chessprogramming.org/`.

[10] Baza podataka online šahovskih igara: `https://www.ficsgames.org/`.

[11] Git repozitorij projektnog zadatka: `https://gitlab.com/filipvukovic/chessrnn`.

Sažetak

U ovom radu bavimo se kreiranjem neuronskog modela za predviđanje ishoda šahovske igre. Za razumijevanje neuronskih modela, prvo definiramo jednostavne neuronske mreže te uvodimo različite varijacije i metode korištene u području strojnog učenja. Potom, definiramo rekurentne neuronske mreže i procese potrebne za učenje takvih mreža. Osnovnu strukturu rekurentnih mreža nadograđujemo s LSTM arhitekturom potrebnu za definiranje modela projektnog zadatka. Nadalje, prikazujemo način zapisivanja šahovskih igara u standardnoj notaciji i uvodimo dva načina prikazivanja stanja šahovske ploče pogodnih za korištenje u neuronskim modelima. Na kraju, definiramo model temeljen na LSTM arhitekturi koji obrađuje šahovsku igru putem niza poteza i uspoređujemo rezultate modela u odnosu na dvije reprezentacije šahovske ploče.

Ključne riječi

strojno učenje, neuronske mreže, rekurentne neuronske mreže, LSTM, šah

Chess game outcome prediction based on recurrent neural networks

Summary

In this paper, we focus on creating a neural model for predicting the outcome of a game of chess. To understand neural models, we first define simple neural networks and introduce various variations and methods used in the field of machine learning. Then, we define recurrent neural networks and the processes required for learning such networks. We enhance the basic structure of recurrent networks with the LSTM architecture necessary for defining the project task model. Furthermore, we show how to record chess games in standard notation and introduce two ways of representing the state of the chessboard suitable for use in neural models. Finally, we define a model based on the LSTM architecture that processes a chess game through a sequence of moves and compare the results of the model with respect to two representations of the chessboard.

Keywords

machine learning, neural networks, recurrent neural networks, LSTM, chess

Životopis

Rođen sam u Osijeku 1998. godine. Srednjoškolsko obrazovanje završavam 2016. godine u I. gimnaziji Osijek te potom upisujem sveučilišni preddiplomski studij Matematike na Odjelu za matematiku na Sveučilištu Josipa Jurja Strossmayera u Osijeku. Po završetku preddiplomskog studija 2020. godine upisujem sveučilišni studij Matematike, smjer: Matematika i računarstvo na istom odjelu.