Primjena rekurentnih neuronskih mreža u analizi programskog koda

Jovanović, Antonio

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:855929

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-20**

Repository / Repozitorij:

Repository of School of Applied Mathematics and Computer Science

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike

Antonio Jovanović

Primjena rekurentnih neuronskih mreža u analizi programskog koda

Diplomski rad

Osijek, 2021.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike

Antonio Jovanović

Primjena rekurentnih neuronskih mreža u analizi programskog koda

Diplomski rad

Mentor: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2021.

Sadržaj

1 Osnove neuronskih mreža **2**
1.1 Neuroni i feedforward neuronske mreže 2
1.2 Aktivacijske funkcije 4
1.3 Učenje neuronskih mreža 5
1.3.1 Algoritam povratne propagacije 7

2 Rekurentne neuronske mreže **11**
2.1 Dinamički sustavi, ciklusi i dijeljenje težina 11
2.2 Učenje rekurentnih neuronskih mreža 13
2.3 LSTM arhitektura 15
2.4 Enkoder-dekoder arhitekture i mehanizam pažnje 18
2.4.1 Bahdanau pažnja 18
2.4.2 Ostali oblici pažnje 20

3 Klasifikacija programskog koda **21**
3.1 code2seq 22
3.1.1 Apstraktno sintaksno stablo 22
3.1.2 Opis modela 22
3.1.3 Treniranje i rezultati 23
3.2 Neural Code Comprehension 24
3.2.1 Opis modela 24
3.2.2 Implementacija i rezultati 25

Uvod

Cilj strojnog učenja je dizajnirati algoritam kojim računalo može nešto naučiti na temelju danih podataka. Što i kako će naučiti ovisi o problemu koji želimo riješiti i dizajnu algoritma. Neki od takvih problema su:

- **Klasifikacija**, gdje želimo odrediti funkciju $f^* : \mathbb{R}^n \to \{1, \ldots, k\}$ koja podatke $\boldsymbol{x} \in \mathbb{R}^n$ pridružuje jednoj od k ponuđenih klasa. Primjerice, podatak može biti vektorska reprezentacija rečenice iz prirodnog jezika, a klase pozitivan i negativan sentiment rečenica.

- **Regresija**, gdje za ulaz $\boldsymbol{x} \in \mathbb{R}^n$ određujemo pripadnu vrijednost $y \in \mathbb{R}$, odnosno želimo odrediti funkciju $f^* : \mathbb{R}^n \to \mathbb{R}$ koja najbolje opisuje dani skup podataka. Jednostavan model regresije je linearna regresija, gdje je model funkcija zadana s $f^*(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$, pri čemu su $\boldsymbol{w} \in \mathbb{R}^n$ i $b \in \mathbb{R}$ parametri koje algoritam treba izračunati.

- **Prijevod teksta**, gdje tekst u nekom prirodnom jeziku prevodimo u neki drugi jezik tako da tekstovi imaju što bliže značenje.

- **Procjena gustoće**, gdje za skup podataka $\{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^N\} \in \mathbb{R}^n$ određujemo funkciju $p_{\text{model}} : \mathbb{R}^n \to \mathbb{R}$ koju interpretiramo kao funkciju gustoće iz koje je skup podataka uzorkovan.

Umjetne neuronske mreže su načini računanja funkcija koje aproksimiramo u gore navedenim problemima, inspirirane radom neurona u ljudskom mozgu. Opis računanja neke funkcije se u teoriji izračunjivosti zove **model računanja** pa možemo reći da su umjetne neuronske mreže specifičan model računanja funkcija.

U ovom radu ćemo se baviti rekurentnim neuronskim mrežama i njihovim primjenama u klasifikaciji, predviđanju naziva i predviđanju funkcionalnosti programskog koda. Ovo polje učenja je znatno napredovalo u zadnjih nekoliko godina, toliko da postoje modeli koji na temelju kratkog opisa funkcije mogu samostalno i točno napisati tijelo te funkcije.[1]

U prvom poglavlju ćemo definirati feedforward neuronske mreže[2] i dati efikasan algoritam za njihovo učenje. U drugom poglavlju ćemo definirati rekurentne neuronske mreže, usporediti ih sa feedforward te izvesti izmijenjeni algoritam za njihovo učenje. Opisat ćemo enkoder-dekoder arhitekturu kojom je moguće definirati generativne (primarno jezične) modele i vidjeti kako riješiti problem dugih ulaznih nizova koristeći tzv. attention mehanizam. Na kraju ćemo dati pregled radova koji koriste rekurentne neuronske mreže za klasifikaciju, predviđanje naziva i funkcionalnosti programskog koda, zajedno sa implementacijom modela za klasifikaciju koda.

[1] Github Copilot (`https://copilot.github.com/`) je modificirana verzija jezičnog GPT-3 modela, namijenjena za pisanje programskog koda.

[2] Izraz je moguće prevesti kao *unaprijedna* neuronska mreža. U nastavku ćemo koristiti *feedforward*

1 Osnove neuronskih mreža

Princip rada umjetnih neuronskih mreža dolazi iz pojednostavljenog poimanja računanja u ljudskom mozgu. Umjesto direktnog računanja složenih izraza, računanje se vrši unutar velikog broja jednostavnih jedinica računanja (neurona) povezanih s ostalim neuronima.

Prvu neuronsku mrežu koju je moguće trenirati, **perceptron**, definirao je Rosenblatt 1957. godine [14]. Ipak, ova grana strojnog učenja pravi procvat je doživjela tek početkom 21. stoljeća.

U ovom poglavlju ćemo definirati feedforward neuronsku mrežu i pokazati kako naučiti neuronsku mrežu da riješi neke od zadataka strojnog učenja.

1.1 Neuroni i feedforward neuronske mreže

Umjetni neuron je funkcija koja računa težinsku sumu n vrijednosti i na nju primijenu neku **aktivacijsku funkciju** $a : \mathbb{R} \to \mathbb{R}$. Drugim riječima, za ulaze $x_1, \ldots, x_n \in \mathbb{R}$ i težine $w_1, \ldots, w_n \in \mathbb{R}$ neuron računa

$$h := a(w_1 x_1 + \cdots + w_n x_n + b), \tag{1.1}$$

gdje je b tzv. *bias parametar*[3]. Kombinirajući neurone, težine veza između neurona i različite aktivacijske funkcije moguće je složiti neuronsku mrežu.

Primjer 1.1. Pretpostavimo da smo za rješavanje problema klasifikacije s dvije klase za model funkciju odabrali

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{W}_2^\intercal \tanh(\boldsymbol{W}_1^\intercal \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2), \tag{1.2}$$

gdje je $\boldsymbol{x} \in \mathbb{R}^3, \boldsymbol{W}_1 \in \mathbb{R}^{3\times 4}, \boldsymbol{W}_2 \in \mathbb{R}^{4\times 2}, \boldsymbol{b}_1 \in \mathbb{R}^4, \boldsymbol{b}_2 \in \mathbb{R}^2$, a funkcije $\sigma : \mathbb{R} \to (0,1)$ sigmoidalna funkcija zadana s $\sigma(x) = \frac{1}{1+\exp(-x)}$ i $\tanh : \mathbb{R} \to (-1,1)$ tangens hiperbolni koje se na vektore primijenjuju po komponentama.

Izraz je nezgrapan, ali ga je moguće prikazati kao kompoziciju jednostavnijih funkcija i prikazati skupom povezanih neurona na slici 1.

Prvi, lijevi sloj mreže nazivamo ulaznim slojem, desni sloj mreže zovemo izlaznim slojem, dok sve slojeve između ulaznog i izlaznog zovemo skrivenim slojevima mreže. Sve težine između i-tog i $(i+1)$-vog sloja prikazujemo matricom $\boldsymbol{W}_i$. Primjerice, težina između neurona x_2 i h_3 jednaka je $\boldsymbol{W}_{1_{3,2}}$.

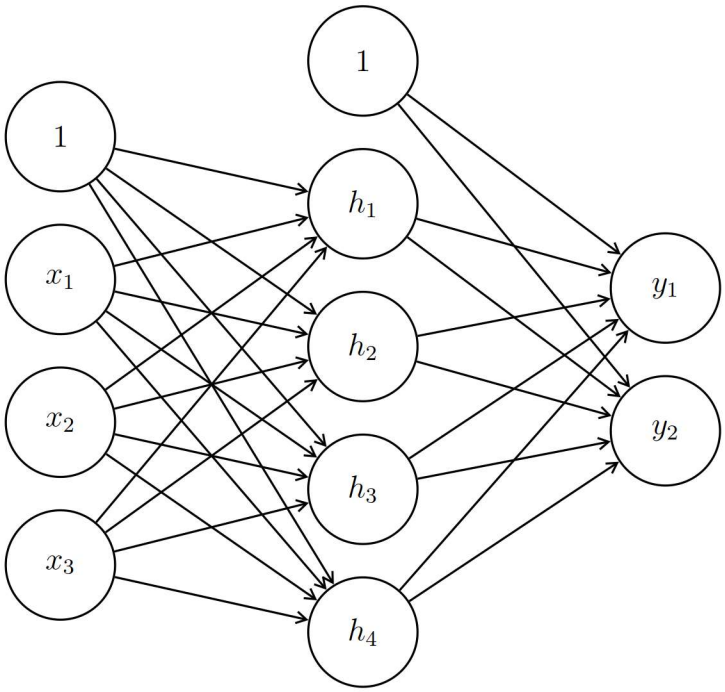
Uočimo da se neke oznake u grafičkom prikazu razlikuju od onih u izrazu (1.2). Umjesto oznake $\boldsymbol{b}$ za bias parametar koristimo neuron čija je vrijednost uvijek jednaka 1, a elemente vektora $\boldsymbol{b}$ postavljamo kao težine bridova koji spajaju taj bias neuron s neuronima u sljedećem sloju.

Na ovaj način smo funkciju f prikazali kao $f = f_2 \circ f_1$, gdje su

$$\begin{aligned} f_1(\boldsymbol{x}) &= \tanh(\boldsymbol{W}_1^\intercal \boldsymbol{x} + \boldsymbol{b}_1) \\ f_2(\boldsymbol{h}) &= \sigma(\boldsymbol{W}_2^\intercal \boldsymbol{h} + \boldsymbol{b}_2). \end{aligned}$$

Iako za shvaćanje rada neuronskih mreža nije nužno dati strogu definiciju, korisno je zbog jednostavnijeg rješavanja eventualnih nedoumica.

[3]Može se koristiti naziv *prag*, jer određuje koliku vrijednost težinska suma mora postići da bi rezultat bio veći od nule. U nastavku ćemo koristiti izraz *bias parametar*.



Slika 1: Arhitektura feedforward neuronske mreže s jednim skrivenim slojem

Definicija 1.1. *Feedforward neuronska mreža je uređena trojka* (G, w, A)*, gdje je* $G = (V, E)$ *usmjeren, aciklički, povezan graf,* $w : E \to \mathbb{R}$ *funkcija težine nad bridovima i* $A : V \to \mathbb{R}^{\mathbb{R}}$ *funkcija koja svakom vrhu pridružuje aktivacijsku funkciju* $a_v := A(v)$*.*

Definicija 1.1 ništa ne govori o računanju unutar neuronske mreže, ali za računanje želimo iskoristiti definiciju neurona (1.1). Svakom neuronu bez ulaznih bridova dodijeljujemo komponentu ulaza u mrežu $\boldsymbol{x}$ ili vrijednost 1 ako se radi o bias neuronu te sve takve vrhove označavamo posjećenima. Zbog konstrukcije grafa G, sve dok svi vrhovi nisu posjećeni, možemo odabrati jedan vrh v takav da su svi vrhovi u za koje postoji brid $(u, v) \in E$ posjećeni, te izračunati

$$z_v := \sum_{(u,v)\in E} w(u, v)h_u \tag{1.3}$$

$$h_v := a_v(z_v). \tag{1.4}$$

Vrijednosti neurona koji nemaju izlaznih bridova tada smatramo izlazima mreže.

Nadalje ćemo se ograničiti samo na specifičan podskup feedforward neuronskih mreža. Kao u primjeru 1.1, poželjno je neurone razvrstati u *slojeve*, odnosno napraviti particiju skupa V na neprazne, disjunktne podskupove $V_0, V_1, V_2, \ldots, V_L$ koji u uniji daju V. Particiju radimo tako da svaki brid $e \in E$ povezuje vrhove iz V_{i-1} i V_i za neki $i \in \{1, \ldots, L\}$.

Unutar svakog sloja osim zadnjeg postavljamo bias neuron čija vrijednost uvijek iznosi 1, odnosno aktivacijska funkcija mu je zadana s $a_v(x) = 1$.

Skup V_0 nazivamo ulaznim slojem, V_L izlaznim, a $V_1, \ldots, V_{L-1}$ skrivenim slojevima mreže. Broj L nazivamo dubinom mreže, a $\max_{l=0,1,\ldots,L} |V_l|$ širinom mreže.

Kako u neuronskoj mreži za ulaz $\boldsymbol{x} \in \mathbb{R}^n$ želimo izračunati neki izlaz $\hat{\boldsymbol{y}} \in \mathbb{R}^m$, neuronsku mrežu konstruiramo tako da vrijedi $n = |V_0| - 1$ (zbog bias neurona) i $m = |V_L|$.

1.2 Aktivacijske funkcije

Nakon što se unutar neurona izračuna težinska suma vrijednosti neurona iz prethodnog sloja, na tu vrijednost djelujemo nekom funkcijom $a : \mathbb{R} \to \mathbb{R}$. U modelima strojnog učenja gotovo uvijek rješavamo nekakav problem optimizacije pa je poželjno da takve funkcije budu neprekidne i diferencijabilne. Neke od aktivacijskih funkcija koje se često koriste u konstrukciji neuronskih mreža su sljedeće:

- **Identiteta**, zadana s
$$\mathrm{id}(x) = x, \tag{1.5}$$
se gotovo uvijek koristi samo u ulaznom sloju, što znači da za ulaz $(x_1, \ldots, x_n) \in \mathbb{R}^n$ vrijednost u tim neuronima bude $\mathrm{id}(x_i) = x_i$.

  Kada bi svaka aktivacijska funkcija u neuronskoj mreži bila identiteta, model funkcija definirana mrežom bila bi afino preslikavanje oblika $\boldsymbol{x} \mapsto \boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{b}$. Primjerice, neka je zadana mreža kao u primjeru 1.1, ali tako da su sve aktivacije identitete. Tada bi model funkcija bila jednaka
$$\begin{aligned} f(\boldsymbol{x}) &= \mathrm{id}(\boldsymbol{W}_2^\intercal \mathrm{id}(\boldsymbol{W}_1^\intercal \boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b}_2) \\ &= \boldsymbol{W}_2^\intercal \boldsymbol{W}_1^\intercal \boldsymbol{x} + \boldsymbol{W}_2^\intercal \boldsymbol{b}_1 + \boldsymbol{b}_2, \end{aligned}$$
što je točno oblika $\boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{b}$. Zato, ukoliko u neuronsku mrežu ne uvedemo nelinearnost kroz aktivacije, ne možemo riješiti problem klasifikacije s dvije klase za podatke koji nisu linearno separabilni u $\mathbb{R}^n$ (vidi [7, poglavlje 6.1]).

- **ReLU funkcija**, zadana s
$$\mathrm{ReLU}(x) = \max(0, x), \tag{1.6}$$
je po dijelovima linearna funkcija koja se koristi zbog jednostavnosti optimizacija. Iako nije diferencijabilna, moguće je izračunati derivaciju svuda osim u $x = 0$. Kako je $\mathrm{ReLU}''(x) = 0$ za sve $x \in \mathbb{R} \setminus \{0\}$, funkcija je beskorisna u optimizacijskim metodama koje koriste drugu derivaciju.

  Također, funkcija je problematična kada vrijednost većeg broja neurona bude negativna zbog gubitka informacija iz tih neurona.

- **Leaky ReLU funkcija**, zadana s
$$\mathrm{LReLU}(x) = \max(0, x) + \min(0, 0.01x), \tag{1.7}$$
koja rješava problem odumiranja neurona prilikom korištenja ReLU aktivacije, ali ne rješava problem računanja gradijenta u $x = 0$.

- **Softplus funkcija**, zadana s
$$\zeta(x) = \log(1 + \exp(x)), \tag{1.8}$$
predstavlja glatku verziju ReLU funkcije. Iako se čini da je ova funkcija uvijek bolji odabir od ReLU, Glorot et. al [5] tvrde suprotno, odnosno da ReLU daje bolje rezultate od softplus funkcije.

- **Sigmoidna funkcija**, zadana s

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

se koristi prilikom predviđanja vjerojatnosti u problemu klasifikacije s dvije klase jer vraća vrijednost iz intervala $(0, 1)$. Funkcija je diferencijabilna i vrijedi

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)). \tag{1.10}$$

- **Tangens hiperbolni**, zadan s

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} = 2\sigma(2x) - 1, \tag{1.11}$$

je skalirana i translatirana verzija sigmoidne funkcije pa je diferencijabilna:

$$\tanh'(x) = 1 - \tanh^2(x). \tag{1.12}$$

Goodfellow et. al [7, str. 195] predlažu korištenje ove funkcije nad sigmoidnom zbog brže konvergencije učenja.

- **Hard tanh**, zadan s

$$f(x) = \max(-1, \min(1, x)), \tag{1.13}$$

predstavlja "jeftiniju" varijantu tanh funkcije koja daje dobre rezultate samo za $|x| < 1$.

Još jedna važna funkcija je softmax : $\mathbb{R}^n \to \mathbb{R}^n$, zadana s

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}, \quad i = 1, \ldots, n. \tag{1.14}$$

Ova funkcija normalizira vektor $\boldsymbol{x} \in \mathbb{R}^n$ te se može koristiti u klasifikaciji s više klasa. Uočimo da softmax nije aktivacijska funkcija, nego djeluje na cijeli sloj neurona te se često primjenjuje na vrijednosti izlaznog sloja mreže.

Na slici 2 se nalaze grafovi svih navedenih aktivacijskih funkcija i njihovih derivacija.

1.3 Učenje neuronskih mreža

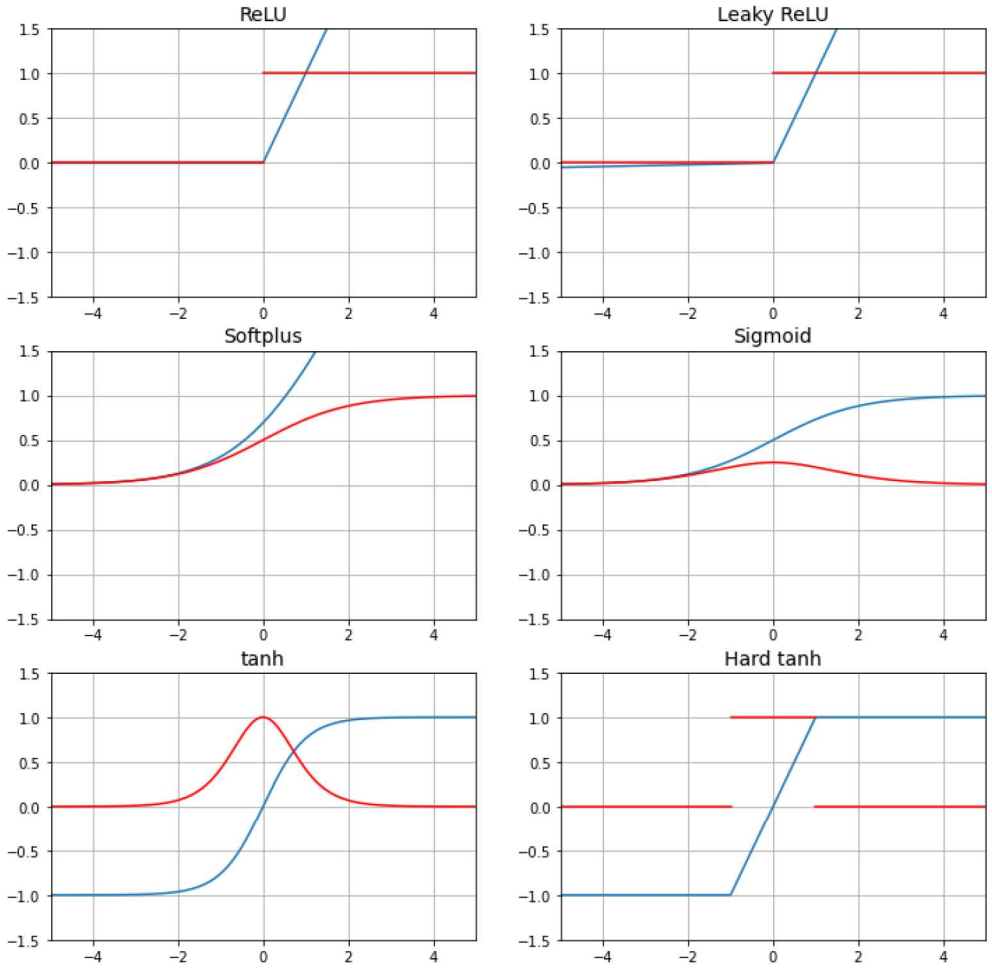
Definiranjem neuronskih mreža smo pojednostavili prikaz kompliciranih model funkcija, ali takav model je beskoristan ukoliko ne možemo efikasno riješiti problem učenja.

Po definiciji 1.1, neuronska mreža se sastoji od usmjerenog acikličkog grafa G, funkcije težine w i aktivacija A. **Arhitekturom mreže** nazivamo uređeni par (G, A), a cilj učenja neuronske mreže je odrediti težine w tako da mreža (G, w, A) što bolje opisuje neki skup podataka te da dobro opisuje neviđene podatke. Iako smo w definirali kao funkciju, možemo je interpretirati kao parametar $\boldsymbol{w} \in \mathbb{R}^k$, gdje graf G sadrži k bridova. Takvo označavanje će nam pojednostaviti algoritam učenja neuronske mreže.

Problemi koje najčešće rješavamo neuronskim mrežama su problemi **nadziranog učenja**, gdje svaki ulazni podatak na kojem treniramo mrežu $\boldsymbol{x} \in \mathbb{R}^n$ ima oznaku $\boldsymbol{y} \in \mathbb{R}^m$, dok mreža proizvodi vrijednost

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}; w).$$

Skup podataka za treniranje označit ćemo kao skup N uređenih parova, $\{(\boldsymbol{x}_1, \boldsymbol{y_1}), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$.



Slika 2: Grafovi aktivacijskih funkcija (plavo) i pripadnih derivacija (crveno).

Prvo je potrebno definirati funkciju

$$\text{loss}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{1.15}$$

koja računa odstupanje točne oznake $\boldsymbol{y}$ od izračunate oznake $\hat{\boldsymbol{y}}$. Funkciju gubitka tada možemo definirati kao prosjek odstupanja po cijelom skupu za treniranje, koji ovisi o parametrima mreže:

$$J(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} \text{loss}(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i). \tag{1.16}$$

Primjer 1.2. Izlazni sloj neuronske mreže koja rješava problem regresije sastoji se od jednog neurona. Grešku modela, ovisnu o skupu podataka $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$ i težinama $\boldsymbol{w}$, možemo definirati kao sumu kvadratnih odstupanja

$$J(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - f(\boldsymbol{x}_i; \boldsymbol{w}))^2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2. \tag{1.17}$$

Diferencijabilnost funkcije J ovisi o diferencijabilnosti model funkcije f.

Primjer 1.3. U problemu klasifikacije s dvije klase zadnji sloj neuronske mreže sadrži jedan neuron s sigmoidnom aktivacijskom funkcijom i predstavlja vjerojatnost da je oznaka jednaka

1. Za definiranje greške modela možemo iskoristiti tzv. unakrsnu entropiju[4]:

$$J(\boldsymbol{w}) = -\frac{1}{N}\sum_{i=1}^{N}\left(\hat{y}_i \log\left(y_i\right) + \left(1-\hat{y}_i\right)\log\left(1-y_i\right)\right). \tag{1.18}$$

Više o statističkom pristupu modeliranju i principu maksimalne vjerodostojnosti, odakle se izvodi izraz (1.18), može se pronaći u [7, str. 131-135]

Ostaje nam konstruirati optimizacijsku metodu kojom se minimizira vrijednost funkcije gubitka. Ako smo vodili brigu da model funkcija i funkcija gubitka budu diferencijabilne, možemo primijeniti neku iterativnu metodu za optimizaciju.

Dobar izbor je metoda gradijentnog spusta, gdje se za funkciju $J : \mathbb{R}^k \to \mathbb{R}$ i polaznu točku $\boldsymbol{w}^{(0)} \in \mathbb{R}^k$ krećemo u smjeru negativnog gradijenta funkcije J u točki $\boldsymbol{w}^{(i)}$, odnosno

$$\boldsymbol{w}^{(i+1)} := \boldsymbol{w}^{(i)} - \lambda \nabla_{\boldsymbol{w}} J(\boldsymbol{w}^{(i)}), \tag{1.19}$$

sve dok se dovoljno ne približimo točki lokalnog minimuma. Uz određene uvjete na funkciju J i odabir duljine koraka λ može se pokazati da gradijentna metoda konvergira točki lokalnog ili globalnog minimuma [16].

Zbog velikog skupa podataka, skupo je u svakoj iteraciji računati cijeli gradijent te se koriste modifikacije gradijentnog spusta:

- Stohastički gradijentni spust, gdje se gradijent aproksimira koristeći samo jedan slučajno odabrani podatak.
- Mini-batch gradijentni spust, gdje se se skup za treniranje podijeli na disjunktne podskupove, te se za svaki takav podskup izračuna gradijent i ažurira parametar. Jedna takva iteracija kroz cijeli skup zove se **epoha** treniranja.

U praksi se često koriste sofisticiranije varijante metode gradijentnog spusta. Naime, u svakoj iteraciji stohastičkog ili mini-batch gradijentnog spusta potrebno je ručno izračunati duljinu koraka. Ukoliko automatiziramo računanje duljine koraka ili procijenimo ponašanje pojedine komponente gradijenta, moguće je znatno ubrzati treniranje modela.

Jedan od popularnijih algoritama je **Adam** [9], skraćeno od *adaptive moment estimation*. Umjesto samog gradijenta, računaju se procjene prvog i drugog momenta gradijenta zbog procjene oblika funkcije gubitka, što dovodi do brže konvergencije u usporedbi sa standardnim metodama gradijentnog spusta. Pregled i usporedba ostalih modifikacija metode gradijentnog spusta može se pronaći u [15].

Drugi legitiman izbor optimizacijskog algoritma je Newtonova metoda ili neke varijante Quasi-Newtonovih metoda. Međutim, Newtonova metoda zahtjeva dvostruku diferencijabilnost funkcije J te se loše ponaša kada Hesijan funkcije nije pozitivno definitan u svakoj iteraciji. Također, Newtonova metoda zahtjeva invertiranje matrice dimenzije $k \times k$, što nije efikasno u slučaju velikog broja parametara k. [7, str. 310-313]

1.3.1 Algoritam povratne propagacije

Uvedimo prvo oznake za težine mreže i vrijednosti neurona. Za $l \in \{1, \ldots, L\}$ težinu između i-tog neurona $(l-1)$-vog sloja i j-tog neurona l-tog sloja označavamo s w_{ij}^l. Iako se težine koje izlaze iz bias neurona nalaze u $\boldsymbol{w}$, bias koji ulazi u i-ti neuron l-tog sloja označavamo

[4] engl. cross-entropy function

Algorithm 1 Optimizacijski algoritam Adam

Require: λ, duljina koraka
Require: $\beta_1, \beta_2 \in [0, 1)$, stope opadanja momenata
Require: $\epsilon \in (0, 10^{-6}]$, konstanta za numeričku stabilnost
Require: J, funkcija gubitka
Require: $\boldsymbol{w}^{(0)}$, početni parametar funkcije gubitka
  $\boldsymbol{m}^{(0)} \leftarrow \mathbf{0}$ (prva aproksimacija prvog momenta jednaka nuli)
  $\boldsymbol{v}^{(0)} \leftarrow \mathbf{0}$ (prva aproksimacija drugog momenta jednaka nuli)
  $t \leftarrow 0$, (iteracija algoritma)
  while $\boldsymbol{w}$ nije dovoljno blizu lokalnom minimumu **do**
    $t \leftarrow t + 1$
    $\boldsymbol{g}^{(t)} \leftarrow \nabla_{\boldsymbol{w}} J(\boldsymbol{w}^{(t-1)})$, (gradijent)
    $\boldsymbol{m}^{(t)} \leftarrow \beta_1 \boldsymbol{m}^{(t-1)} + (1 - \beta_1)\boldsymbol{g}^{(t)}$, (prvi moment)
    $\boldsymbol{v}^{(t)} \leftarrow \beta_2 \boldsymbol{v}^{(t-1)} + (1 - \beta_2)\boldsymbol{g}^{(t)^2}$, (drugi moment)
    $\hat{\boldsymbol{m}}^{(t)} \leftarrow \frac{\boldsymbol{m}^{(t)}}{1-\beta_1^t}$, (korekcija prvog momenta)
    $\hat{\boldsymbol{v}}^{(t)} \leftarrow \frac{\boldsymbol{v}^{(t)}}{1-\beta_2^t}$, (korekcija drugog momenta)
    $\boldsymbol{w}^{(t)} \leftarrow \boldsymbol{w}^{(t-1)} - \lambda \frac{\hat{\boldsymbol{m}}^{(t)}}{\sqrt{\hat{\boldsymbol{v}}^{(t)}} + \epsilon}$
  end while

s b_i^l. Nadalje ćemo pretpostaviti da su u svakom sloju sve aktivacijske funkcije jednake, odnosno

$$a_i^l = a_j^l, \forall i, j \in \{1, \ldots, |V_l|\}.$$

Vrijednost j-tog neurona u l-tom sloju nakon djelovanja aktivacijske funkcije označavamo s

$$h_j^l = a^l\left(\sum_{i=1}^{|V_{l-1}|} w_{ij}^l h_i^{l-1} + b_j^l\right),$$

dok istu vrijednost prije djelovanja aktivacijske funkcije označavamo s z_j^l.

Za računanje gradijenta je za svaki sloj mreže $l \in \{1, \ldots, L\}$ potrebno izračunati

$$\frac{\partial J}{\partial w_{ij}^l}, \forall i, j.$$

Zašto samo ne bismo redom naivno izračunali sve parcijalne derivacije? Tada bi, u transponiranom grafu G', za svaki neuron morali proći svim putevima iz izlaznog sloja do tog neurona. Međutim, za neuron u l-tom sloju takvih puteva postoji $|V_{l+1}| \ldots |V_L|$ pa za velik broj parametara i velik broj slojeva mreže takav pristup nije nimalo efikasan.

Uočimo da u naivnom pristupu velik broj putova dijeli jednake podputove. Kako bismo smanjili broj računanja, grešku l-tog sloja $\boldsymbol{\delta}^l$ definiranu s

$$\delta_j^l := \frac{\partial J}{\partial z_j^l}, \quad j \in \{1, \ldots, |V_l|\} \tag{1.20}$$

ćemo koristiti prilikom računanja grešaka prethodnih slojeva te tako izbjeći višestruko računanje istog izraza.

Koristeći pravilo deriviranja kompozicije funkcija, za izlazni sloj mreže dobivamo

$$\delta_j^L - \frac{\partial J}{\partial z_j^L} - \sum_{i=1}^{|V_L|} \frac{\partial J}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_j^L} - \frac{\partial J}{\partial \hat{y}_j} a^{L'}(z_j^L), \tag{1.21}$$

jer je $\frac{\partial \hat{y}_i}{\partial z_j^L} = 0$ za $i \neq j$ i $\hat{y}_j = a^{L'}(z_j^L)$. Izračunajmo sada greške svih ostalih slojeva.

Propozicija 1.1. *Za sve* $l \in \{0, \ldots, L-1\}$ *i* $j \in \{1, \ldots, |V_l|\}$ *vrijedi*

$$\delta_j^l = \sum_{i=1}^{|V_{l+1}|} w_{ji}^{l+1} \delta_i^{l+1} a^{l'}(z_j^l). \tag{1.22}$$

Dokaz. Kao u (1.21), koristeći pravilo deriviranja kompozicije funkcija dobivamo

$$\delta_j^l = \frac{\partial J}{\partial z_j^l} = \sum_{i=1}^{|V_{l+1}|} \frac{\partial J}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial z_j^l}. \tag{1.23}$$

Kako je

$$z_i^{l+1} = \sum_{k=1}^{|V_l|} w_{ki}^{l+1} a^{l+1}(z_k^l) + b_i^{l+1},$$

deriviranjem dobivamo

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

te uvrštavanjem u (1.23) dobivamo

$$\delta_j^l = \sum_{i=1}^{|V_{l+1}|} \delta_i^{l+1} w_{ji}^{l+1} a^{l+1'}(z_j^l) = \sum_{i=1}^{|V_{l+1}|} w_{ji}^{l+1} \delta_i^{l+1} a^{l+1'}(z_j^l),$$

što je trebalo pokazati. □

Izraz (1.22) je nezgrapan, ali se može vektorski zapisati kao

$$\boldsymbol{\delta}^l = \boldsymbol{W}^{l+1} \boldsymbol{\delta}^{l+1} \odot a^{l'}(\boldsymbol{z}^l), \tag{1.24}$$

gdje je $\boldsymbol{z}^l = (z_1^l, \ldots, z_{|V_l|}^l)^\intercal$, $\boldsymbol{W}^{l+1} \in \mathbb{R}^{|V_l| \times |V_{l+1}|}$ matrica težina između l-tog i $(l+1)$-vog sloja, te $\odot$ oznaka za Hadamardov produkt.

Sada želimo izračunati tražene parcijalne derivacije koristeći izračunate pogreške u neuronima. Ponovnom primijenom pravila derivacije kompozicije dobivamo

$$\frac{\partial J}{\partial w_{ij}^l} = h_i^{l-1} \delta_j^l \tag{1.25}$$

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l \tag{1.26}$$

U algoritmu 2 smo zbog jednostavnosti zapisa mreži dali samo jedan ulazni podatak. Jednostavno ga je modificirati i primijeniti neki od navedenih optimizacijskih algoritama, primjerice mini-batch metoda gradijentnog spusta.

Kvaliteta naučenog modela ne ovisi samo o izboru arhitekture mreže i optimizacijskog algoritma. Prilikom učenja, model je moguće prenaučiti[5], tako da točno predviđa vrijednosti na skupu za treniranje, a loše generalizira, odnosno računa loše vrijednosti za podatke koje nije vidio za vrijeme treniranja.

[5] engl. overfit

Algorithm 2 Algoritam povratne propagacije.

Require: L, dubina mreže
Require: $a^l, l \in \{1, \ldots, L\}$, aktivacijska funkcija za svaki sloj mreže
Require: $\boldsymbol{w}$, parametar koji sadrži sve težine i sve bias parametre između slojeva slojeva $\boldsymbol{W}^l, \boldsymbol{b}^l, l \in \{1, \ldots, L\}$
Require: $\boldsymbol{x}$, ulaz u mrežu
Require: $\boldsymbol{y}$, pripadna oznaka
Require: J, funkcija gubitka
  $\boldsymbol{h}^0 \leftarrow \boldsymbol{x}$
  for $l \leftarrow 1, \ldots, L$ **do** (propagacija unaprijed)
    $\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{h}^{l-1} + \boldsymbol{b}^l$
    $\boldsymbol{h}^l = a^l(\boldsymbol{z}^l)$
  end for
  $\hat{\boldsymbol{y}} \leftarrow \boldsymbol{h}^L$
  $J(\boldsymbol{w}) \leftarrow \text{loss}(\boldsymbol{y}, \hat{\boldsymbol{y}})$
  $\boldsymbol{\delta}^L \leftarrow \nabla_{\hat{\boldsymbol{y}}} J \odot a^{L'}(\boldsymbol{z}^L)$
  for $l \leftarrow L-1, \ldots, 1, 0$ **do** (propagacija unatrag)
    $\boldsymbol{\delta}^l \leftarrow \boldsymbol{W}^{l+1} \boldsymbol{\delta}^{l+1} \odot a^{l'}(\boldsymbol{z}^l)$.
  end for
  Ažuriraj parametar $\boldsymbol{w}$ koristeći izračunate vrijednosti $\boldsymbol{\delta}^l$, izraze (1.25) i (1.26).

Za primjer, pretpostavimo da je funkcija koju želimo aproksimirati polinom trećeg stupnja, a skup za treniranje se sastoji od N podataka. Ukoliko za model funkciju odaberemo polinom stupnja N, nakon učenja će funkcija točno opisivati cijeli skup za treniranje. Međutim, ukoliko uvedemo neki novi, neviđeni podatak, cijeli model će se raspasti.

Goodfellow i suradnici [7] regularizaciju definiraju kao modifikaciju algoritma učenja koja smanjuje grešku generalizacije bez da smanji grešku tijekom učenja. Popularnu metodu regularizacije u feedforward i rekurentim mrežama, *dropout*, definirali su Srivastava i suradnici [17] 2014. godine. Ideja je, za vrijeme propagacije unaprijed, vrijednosti nekih slučajno odabranih neurona postaviti na 0 i prestati ih koristiti za daljnje računanje, što tjera mrežu da zaključuje na temelju rijetkih podataka i sprječava prenaučavanje.

2 Rekurentne neuronske mreže

Prilikom treniranja i upotrebe feedforward neuronske mreže, kao ulaz mreži uvijek dajemo jedan podatak kojemu su komponente uglavnom međusobno nezavisne. Što ako želimo obraditi podatak u kojemu komponenta vektora ovisi o svim prethodnim komponentama ili želimo obraditi više podataka koji međusobno ovise jedni o drugima? Pokušajmo riješiti neki od takvih problema koristeći feedforward neuronske mreže.

Primjer 2.1. Želimo dizajnirati neuronsku mrežu koja određuje je li komentar na Facebooku pozitivan ili negativan. Pretpostavimo da svaka riječ ima svoju vektorsku reprezentaciju dimenzije 100[6] (vidi [11]). Jedan pristup bi bio mreži proslijediti jednu po jednu riječ, na izlazu izračunati doprinosi li ta riječ pozitivnom ili negativnom sentimentu te izračunati težinsku sumu svih dobivenih vrijednosti i donijeti odluku. U ovakvom pristupu gubi se zavisnost riječi o njezinoj okolini. Ukoliko mreži proslijedimo komentar

Želim se cijepiti jer ne volim nositi maske.

mreža ne bi mogla naučiti odnosi li se riječ *ne* na želju za cijepljenjem ili na stav prema nošenju maski.

U drugom pristupu, za komentar duljine N, možemo konstruirati mrežu s ulaznim slojem dimenzije $100N$ i kao ulaz proslijediti konkateniranih N vektora. Međutim, za komentare različite duljine tada moramo trenirati različite mreže, što nije isplativo jer je duljina teksta proizvoljna pa je broj parametara koje mreža mora naučiti iznimno velik.

U ovom poglavlju ćemo opisati različite vrste rekurentnih neuronskih mreža, čiji je cilj obraditi niz vrijednosti $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(T)}$[7] koji nisu međusobno nezavisni i pri tome izbjeći probleme iz primjera 2.1.

2.1 Dinamički sustavi, ciklusi i dijeljenje težina

Promotrimo prvo jednostavan dinamički sustav sa stanjem $\boldsymbol{h}^{(t)}$, vanjskim signalom $\boldsymbol{x}^{(t)}$ i parametrom $\boldsymbol{w}$:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{w}). \tag{2.1}$$

Za računanje stanja $\boldsymbol{h}^{(t)}$ jednadžba sustava koristi signal $\boldsymbol{x}^{(t)}$ i prethodno stanje $\boldsymbol{h}^{(t-1)}$ koje sadrži informacije o svim prethodnim signalima. Na sličan način želimo konstruirati arhitekturu rekurentne neuronske mreže.

Za definiciju rekurentne neuronske mreže, umjesto acikličkog grafa G iz definicije 1.1 možemo zahtjevati da postoji brid između svaka dva vrha $u, v \in V$. Kako bi računanje imalo smisla, između vrhova koje ne želimo povezati dovoljno je staviti težinu 0.

Umjesto takvog pristupa, različite vrste rekurentnih neuronskih mreža opisat ćemo kao niz nekakvih preslikavanja. Kako će se mreže uglavnom sastojati od ulaznog, skrivenog i izlaznog sloja, redom za njihove dimenzije uvodimo oznake n, d, m. Za najjednostavniji oblik definiramo matrice težina

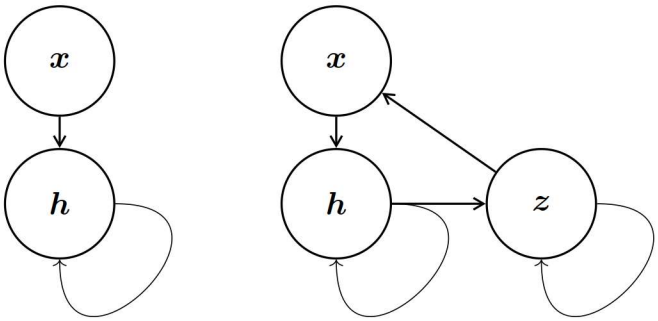
$$\boldsymbol{W}_{xh} \in \mathbb{R}^{d \times n}, \boldsymbol{W}_{hh} \in \mathbb{R}^{d \times d},$$

bias težinu $\boldsymbol{b}_h \in \mathbb{R}^d$, aktivacijsku funkciju $a_h : \mathbb{R} \to \mathbb{R}$, početno skriveno stanje $\boldsymbol{h}^{(0)}$ te rekurentnu neuronsku mrežu kao preslikavanje

$$(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)}) \mapsto (\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \ldots, \boldsymbol{h}^{(T)})$$

[6] Preslikavanje riječi u vektorski prostor zove se ugrađivanje riječi (engl. word embedding).

[7] Oznake t i T uvodimo zbog vremenske zavisnosti u nizu.



Slika 3: Lijevo: Graf kojim je prirodno opisan dinamički sustav (2.1). Vrijednost stanja $\boldsymbol{h}$ se računa koristeći ulaz $\boldsymbol{x}$ (brid iz $\boldsymbol{x}$ u $\boldsymbol{h}$) i prethodno stanje sustava (brid iz $\boldsymbol{h}$ u $\boldsymbol{h}$).
Desno: Graf kojemu je teško definirati djelovanje. Nejasno je kako vrijednost $\boldsymbol{z}$ utječe na ulaz $\boldsymbol{x}$.

definirano s

$$\boldsymbol{z}^{(t)} = \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{b}_h \tag{2.2}$$

$$\boldsymbol{h}^{(t)} = a_h(\boldsymbol{z}^{(t)}). \tag{2.3}$$

Matrica $\boldsymbol{W}_{xh}$ ovdje predstavlja težinu između ulaznog i skrivenog sloja, dok matrica $\boldsymbol{W}_{hh}$ predstavlja težinu kojom skriveni sloj preslikavamo u samog sebe. Glavna razlika između rekurentne i feedforward arhitekture je u ponovnom korištenju istih težina tijekom računanja. Naime, u feedforward mreži svaku matricu težina koristimo točno jednom, za prijelaz između l-tog i $(l+1)$-vog sloja. S druge strane, u rekurentnoj mreži težine $\boldsymbol{W}_{xh}$ i $\boldsymbol{W}_{hh}$ koristimo onoliko puta kolika je duljina ulaznog niza. Zbog dijeljenja težina između koraka je potrebno izmijeniti algoritam povratne propagacije za treniranje mreže, što ćemo napraviti u poglavlju 2.2.

Očito ovo nije jedina vrsta rekurentne neuronske mreže. Primjerice, u problemu određivanja vrste riječi u rečenici, za svaku riječ $\boldsymbol{x}^{(t)}$ možemo računati klasu $\hat{\boldsymbol{y}}^{(t)}$ koristeći skriveno stanje $\boldsymbol{h}^{(t)}$ i matricu težina $\boldsymbol{W}_{hy}$ između skrivenog i izlaznog sloja te bias težinu $\boldsymbol{b}_y$ i aktivacijsku funkciju a_y. Uz izraze (2.2) i (2.3) još bismo računali

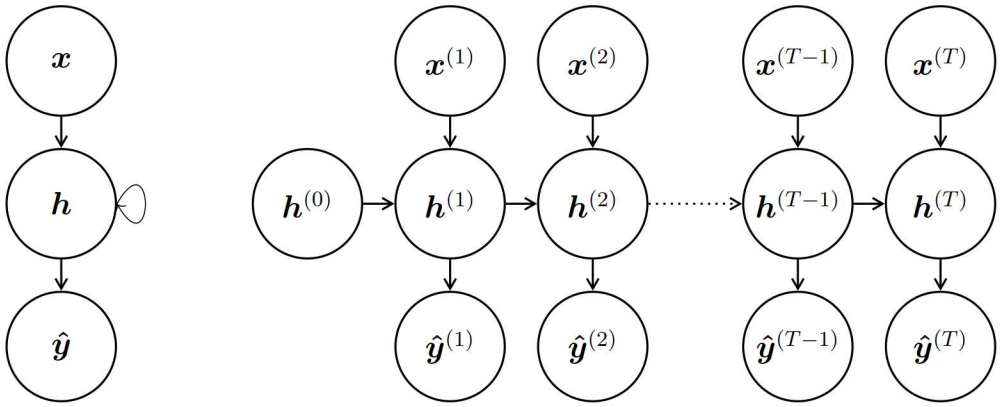
$$\hat{\boldsymbol{y}}^{(\boldsymbol{t})} = a_y(\boldsymbol{W}_{hy}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y). \tag{2.4}$$

Dosad smo samo opisali probleme koji preslikavaju ulazni niz u izlazni niz iste duljine. Još neki problemi koje možemo modelirati rekurentnim neuronskim mrežama su:

- Preslikavanje niza proizvoljne duljine u jednu vrijednost, odnosno niz duljine jedan, se dobije jednostavnom modifikacijom prijašnjih modela. Umjesto da $\hat{\boldsymbol{y}}$ računamo u svakom koraku, dovoljno ga je izračunati samo na kraju, kao

$$\hat{\boldsymbol{y}} = a_y(\boldsymbol{W}_{hy}\boldsymbol{h}^{(T)} + \boldsymbol{b}_y).$$

- Preslikavanje niza fiksne duljine u niz proizvoljne duljine je malo složeniji problem. Zamislimo da su nam dani meteorološki instrumenti koji prikupljaju nekakve podatke svake sekunde. Na temelju tih podataka, želimo stvoriti niz svih vremenskih uvjeta koji su se pojavili u danu. Niz od 86400 mjerenja je potrebno preslikati u niz proizvoljne duljine, primjerice *maglovito, kišovito, oblačno.*



Slika 4: Lijevo: Graf rekurentne neuronske mreže u kojoj se za svako skriveno stanje $\boldsymbol{h}$ računa izlazna vrijednost $\hat{\boldsymbol{y}}$.
Desno: "Odrolani" graf iste neuronske mreže za ulazni niz duljine T.

- Preslikavanje niza proizvoljne duljine u niz proizvoljne duljine. Ovakvu mrežu je potrebno konstruirati ako rješavamo problem prevođenja teksta. Naime, tekst u polaznom jeziku može biti proizvoljne duljine, dok duljina u dolaznom jeziku ne mora biti jednaka onoj u polaznom jeziku. Ideja je konstruirati dvije rekurentne neuronske mreže, jedna koja će polazni tekst preslikati u vektor fiksne duljine, dok druga na temelju tog vektora generira prevedeni tekst. Ovom problemu posvetit ćemo se u poglavlju 2.4.

- Ukoliko element ulaznog niza $\boldsymbol{x}^{(t)}$ ne ovisi samo o prethodnim, nego i o sljedećim elementima, poželjno je računati skrivena stanja unaprijed

$$\overrightarrow{\boldsymbol{h}}^{(t)} = a_h(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$$

  i unazad

$$\overleftarrow{\boldsymbol{h}}^{(t)} = a_h(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$$

  te ih nekako kombinirati, primjerice konkateniranjem:

$$\boldsymbol{h}^{(t)} = \begin{bmatrix} \overrightarrow{\boldsymbol{h}}^{(t)} \\ \overleftarrow{\boldsymbol{h}}^{(t)} \end{bmatrix}. \tag{2.5}$$

  Ovakvu arhitekturu nazivamo **dvosmjernom**.

2.2 Učenje rekurentnih neuronskih mreža

Isto kao kod feedforward neuronskih mreža, potrebno je dati algoritam za treniranje rekurentne neuronske mreže. Kako u grafu rekurentne mreže postoje ciklusi, nije očito kako možemo modificirati algoritam povratne propagacije. Međutim, ukoliko "odrolamo" graf računanja, kao na slici 4, dobivamo usmjereni, aciklički graf. Ukoliko želimo primijeniti povratnu propagaciju za učenje, moramo prvo riješiti problem dijeljenih težina. Naime, u definiciji feedforward neuronskih mreža i algoritmu 2 smo pretpostavili da su težine između svaka dva sloja različite, dok se u rekurentnom slučaju ponavljaju. Pretpostavimo sada da težinu $\boldsymbol{w}$ koristi T različitih vrhova u grafu računanja te da su označene $\boldsymbol{w}^{(1)}, \ldots, \boldsymbol{w}^{(T)}$. Neka

je J funkcija gubitka. Vrijedi:

$$\frac{\partial J}{\partial \boldsymbol{w}} = \sum_{t=1}^{T} \frac{\partial J}{\partial \boldsymbol{w}^{(t)}} \frac{\partial \boldsymbol{w}^{(t)}}{\partial \boldsymbol{w}} \tag{2.6}$$

$$= \sum_{t=1}^{T} \frac{\partial J}{\partial \boldsymbol{w}^{(t)}}. \tag{2.7}$$

Drugim riječima, za računanje gradijenta dijeljenih težina, dovoljno je pretpostaviti da su sve težine nezavisne i nakon propagacije unatrag zbrojiti njihove gradijente.

U nastavku ćemo pretpostaviti da računamo gradijent samo za jedan par $(\boldsymbol{x}, \boldsymbol{y})$ te da modeliramo klasifikator s više klasa odnosno u svakom koraku t na vektor $\hat{\boldsymbol{y}}^{(t)}$ djelujemo softmax funkcijom:

$$\boldsymbol{p}^{(t)} = \text{softmax}(\hat{\boldsymbol{y}}^{(t)}).$$

Prirodno je u svakom koraku računati negativnu log-vjerodostojnost oznake $\boldsymbol{y}^{(t)}$[8] obzirom na dio ulaznog niza $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}$, odnosno računati

$$J(\boldsymbol{w}) = \sum_{t=1}^{T} J^{(t)} \tag{2.8}$$

$$= -\sum_{t=1}^{T} \log p_{\text{model}}(\boldsymbol{y}^{(t)} | \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}), \tag{2.9}$$

gdje s $J^{(t)}$ označavamo gubitak u koraku t, a $p_{\text{model}}(\boldsymbol{y}^{(t)} | \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\})$ računamo kao vrijednost komponente $\boldsymbol{y}^{(t)}$ u normaliziranom vektoru $\boldsymbol{p}^{(t)}$. Algoritam kojim treniramo rekurentne neuronske mreže naziva se **povratna propagacija kroz vrijeme**[9] i sastoji se od sljedećih koraka:

1. Redom mreži dajemo elemente ulaznog niza $\boldsymbol{x}^{(t)}$ i računamo pripadne greške $J^{(t)}$.
2. Za svaki $t \in \{T, \ldots, 1\}$ unazad računamo parcijalne derivacije obzirom na težine:

$$\frac{\partial J}{\partial \boldsymbol{W}_{xh}^{(t)}}, \frac{\partial J}{\partial \boldsymbol{W}_{hh}^{(t)}}, \frac{\partial J}{\partial \boldsymbol{W}_{hy}^{(t)}}, \frac{\partial J}{\partial \boldsymbol{b}_{h}^{(t)}}, \frac{\partial J}{\partial \boldsymbol{b}_{y}^{(t)}}.$$

[8] Oznaku možemo zapisati kao one-hot enkodirani vektor $\boldsymbol{y}^{(t)} \in \mathbb{R}^m$ ili skalar $y^{(t)} \in \{1, \ldots, m\}$.
[9] engl. backpropagation through time, BPTT

3. Zbrajamo sve dijeljene težine:

$$\frac{\partial J}{\partial \boldsymbol{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial J}{\partial \boldsymbol{W}_{xh}^{(t)}},$$
$$\frac{\partial J}{\partial \boldsymbol{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial J}{\partial \boldsymbol{W}_{hh}^{(t)}},$$
$$\frac{\partial J}{\partial \boldsymbol{W}_{hy}} = \sum_{t=1}^{T} \frac{\partial J}{\partial \boldsymbol{W}_{hy}^{(t)}},$$
$$\frac{\partial J}{\partial \boldsymbol{b}_{h}} = \sum_{t=1}^{T} \frac{\partial J}{\partial \boldsymbol{b}_{h}^{(t)}},$$
$$\frac{\partial J}{\partial \boldsymbol{b}_{y}} = \sum_{t=1}^{T} \frac{\partial J}{\partial \boldsymbol{b}_{y}^{(t)}}.$$

Iako algoritmom povratne propagacije kroz vrijeme možemo natrenirati bilo kakvu rekurentnu neuronsku mrežu, u slučaju dugih ulaznih nizova može biti vremenski i prostorno neefikasan. Naime, u grafu računanja je svako skriveno stanje osim zadnjeg povezano sa sljedećim skrivenim stanjem, što daje put u grafu duljine $\mathcal{O}(T)$. Za računanje gradijenta potrebno je čuvati svih $\mathcal{O}(T)$ stanja mreže, što daje veliku memorijsku složenost. Aggarwal [1, str. 281] kao jedno rješenje daje algoritam krnje povratne propagacije kroz vrijeme, gdje umjesto računanja cijelog gradijenta, računa samo dijelove gubitka i gradijenta na podnizovima male duljine, primjerice 100, čime računa aproksimaciju gradijenta.

Goodfellow i suradnici [7, str. 381-384] kao drugo rješenje uvode *teacher forcing*. Umjesto da skriveno stanje $\boldsymbol{h}^{(t)}$ ovisi o $\boldsymbol{x}^{(t)}$ i $\boldsymbol{h}^{(t-1)}$, ono ovisi o $\boldsymbol{x}^{(t)}$ i točnoj oznaci $\boldsymbol{y}^{(t-1)}$ iz prethodnog koraka. U ovakvom pristupu najdulji put u grafu računanja prolazi samo kroz tri vrha te se brže trenira. Prilikom testiranja mreži nije moguće dati točne oznake $\boldsymbol{y}$, nego je potrebno dati izračunate oznake $\hat{\boldsymbol{y}}$. Ukoliko skup podataka za treniranje nije dovoljno reprezentativan, često ovako trenirana mreža ima problema kod generalizacije.

2.3 LSTM arhitektura

Prilikom propagiranja gradijenata u algoritmu povratne propagacije kroz vrijeme nerijetko se javlja problem nestajućeg ili eksplodirajućeg gradijenta, gdje zbog duljine puta u grafu računanja gradijenti teže nuli ili eksponencijalno rastu, što usporava ili onemogućava učenje. Za računanje t-tog skrivenog stanja potrebno je t puta djelovati aktivacijskom funkcijom skrivenog sloja. Pretpostavimo, radi demonstracije, da vrijednost skrivenog sloja ovisi samo o prethodnom skrivenom sloju i da je aktivacijska funkcija identiteta. Tada je rekurzija zadana s

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)}, \tag{2.10}$$

dok parcijalna derivacija funkcije gubitka obzirom na $\boldsymbol{h}^{(t)}$ iznosi

$$\frac{\partial J}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial J}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial J}{\partial \boldsymbol{h}^{(t+1)}} \boldsymbol{W}_{hh} \tag{2.11}$$

Teleskopiranjem do T dobivamo

$$\frac{\partial J}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial J}{\partial \boldsymbol{h}^{(T)}} \boldsymbol{W}_{hh}^{T-t}. \tag{2.12}$$

Ukoliko je $\boldsymbol{W}_{hh} \in \mathbb{R}^{d\times d}$ simetrična, može se zapisati kao

$$\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\intercal}, \tag{2.13}$$

gdje je $\boldsymbol{Q} \in \boldsymbol{R}^{d\times d}$ realna ortogonalna matrica, a $\boldsymbol{\Lambda} \in \mathbb{R}^{d\times d}$ dijagonalna matrica koja na dijagonali sadrži svojstvene vrijednosti od $\boldsymbol{W}_{hh}$ [6, tm. 8.1.1, str. 440]. Uvrštavanjem dekompozicije u teleskopirani izraz dobivamo

$$\frac{\partial J}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial J}{\partial \boldsymbol{h}^{(T)}}\boldsymbol{Q}\boldsymbol{\Lambda}^{T-t}\boldsymbol{Q}^{\intercal}. \tag{2.14}$$

Sve vrijednosti matrice $\boldsymbol{\Lambda}^{T-t}$ manje od 1 isčezavaju za velik $T-t$, dok one veće od 1 eksplodiraju te se isto događa s gradijentima.

Iako smo problem demonstrirali na pojednostavljenom primjeru, u praksi se pojavljuje neovisno o izboru arhitekture i aktivacijskih funkcija, pogotovo ako model mora "uhvatiti" vezu između udaljenih elemenata ulaznog niza. Neke od metoda kojima se izbjegavaju problemi eksplodirajućeg i nestajućeg gradijenta su sljedeće:

- Izrezivanje gradijenta[10], gdje se smanjuju vrijednosti velikih gradijenata. Ukoliko norma gradijenta postane veća od unaprijed zadanog hiperparametra, gradijent se normalizira na vrijednost tog hiperparametra. Ovaj pristup ne rješava problem nestajućeg gradijenta. Za ostale pristupe izrezivanja gradijenta vidi [13].

- Normalizacija sloja[11], gdje se prije djelovanja aktivacijske funkcije na vrijednost skrivenog sloja vrijednost neurona standardizira, skalira i translatira [1, str. 289-290].

- Long short-term memory arhitektura, koju ćemo opisati u nastavku poglavlja.

Jedna ideja je izbaciti matrično množenje kod računanja novog skrivenog stanja i zamijeniti ga zbrajanjem. Rješenje koje su ponudili Hochreiter i Schmidhuber [8] standardni neuron zamjenjuje s tzv. LSTM ćelijom.

Uz skriveno stanje $\boldsymbol{h}^{(t)}$, LSTM ćelija sadrži stanje ćelije $\boldsymbol{c}^{(t)} \in \mathbb{R}^d$ koje služi kao dugoročna memorija tako da kombinira "pamćenje" i "zaboravljanje" informacije iz ranijih slojeva. Primjerice, prilikom obrade teksta mreža može uključiti dugoročno pamćenje kada susretne zagradu ili navodnike, te je isključiti kada susretne zatvarajuću zagradu ili navodnike te tako uhvatiti dugoročne zavisnosti unutar podniza ulazne rečenice.

LSTM arhitektura se može opisati kao niz jednadžbi kojima se računaju $\boldsymbol{h}^{(t)}$ i $\boldsymbol{c}^{(t)}$:

$$\boldsymbol{f}^{(t)} = \sigma(\boldsymbol{W}_f\boldsymbol{x}^{(t)} + \boldsymbol{U}_f\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f) \tag{2.15}$$
$$\boldsymbol{i}^{(t)} = \sigma(\boldsymbol{W}_i\boldsymbol{x}^{(t)} + \boldsymbol{U}_i\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i) \tag{2.16}$$
$$\boldsymbol{o}^{(t)} = \sigma(\boldsymbol{W}_o\boldsymbol{x}^{(t)} + \boldsymbol{U}_o\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o) \tag{2.17}$$
$$\tilde{\boldsymbol{c}}^{(t)} = \tanh(\boldsymbol{W}_c\boldsymbol{x}^{(t)} + \boldsymbol{U}_c\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_c) \tag{2.18}$$
$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \tilde{\boldsymbol{c}}^{(t)} \tag{2.19}$$
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh(\boldsymbol{c}^{(t)}), \tag{2.20}$$

gdje su $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{b}$ parametri koje je potrebno naučiti. Vektore $\boldsymbol{f}, \boldsymbol{i}, \boldsymbol{o}$ i $\tilde{\boldsymbol{c}}$ interpretiramo kao binarne sklopove koji odlučuju kako se neka informacija propagira kroz kratkoročnu ili dugoročnu memoriju:

[10] engl. gradient clipping
[11] engl. layer normalization

- Vektor $\boldsymbol{f}^{(t)} \in (0,1)^k$ (naziv dolazi od engl. forget gate, odnosno sklop za zaboravljanje) odlučuje koliko će se dugoročne informacije iz stanja $\boldsymbol{c}^{(t-1)}$ zaboraviti u stanju $\boldsymbol{c}^{(t)}$, što je vidljivo u prvom pribrojniku izraza (2.19). Primjerice, ukoliko je $f_j^{(t)} = 0$, vrijednost $c_j^{(t-1)}$ se potpuno zaboravlja.

- Vektor $\boldsymbol{i}^{(t)} \in (0,1)^k$ (engl. input gate, ulazni sklop) odlučuje koliko novog ulaza $\tilde{\boldsymbol{c}}^{(t)}$ ulazi u dugoročnu memoriju.

- Vektor $\boldsymbol{o}^{(t)} \in (0,1)^k$ (engl. output gate, izlazni sklop) odlučuje koliko dugoročne memorije ulazi u skriveno stanje $\boldsymbol{h}^{(t)}$.

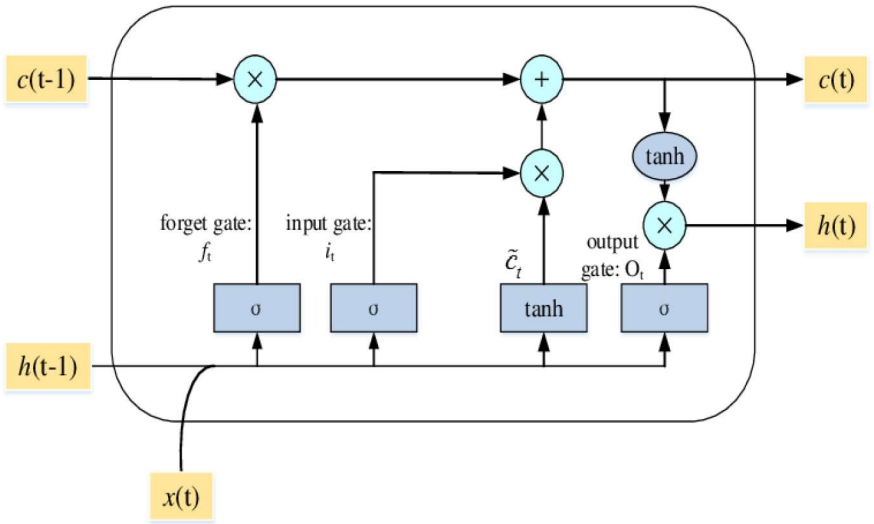
- Vektor $\tilde{\boldsymbol{c}}^{(t)} \in (-1,1)^k$ predstavlja vrijednost novog ulaza, izračunatog iz prethodnog skrivenog stanja $\boldsymbol{x}^{(t-1)}$ i ulaza u mrežu $\boldsymbol{x}^{(t)}$.

Grafički prikaz LSTM ćelije se nalazi na slici 5.

Umjesto množenja kojim se računaju vrijednosti skrivenih stanja u standardnoj rekurentnoj mreži, stanja ćelija se u LSTM arhitekture ažuriraju zbrajanjem, što rezultira boljim protokom gradijenata kroz vremenske korake i izostankom problema nestajućeg gradijenta. Za inicijalizaciju bias parametra $\boldsymbol{b}_f$ u forget sklopu Aggarwal [1, str. 294] preporuča veliku vrijednost zbog boljeg izbjegavanja tog problema.

Često se umjesto standardnog LSTM-a koristi višeslojni LSTM, gdje se u svakom sloju računaju nova skrivena stanja. U prvom sloju se kao početno skriveno stanje mreži daje proizvoljan vektor, često $\mathbf{0}$, dok je početno skriveno stanje l-tog sloja $\boldsymbol{h}_l^{(0)}$ jednako zadnjem stanju prethodnog sloja $\boldsymbol{h}_{l-1}^{(T)}$. Zbog većeg broja slojeva poželjno je koristiti dovoljno velik skup za treniranje kako bi se izbjegla pretreniranost modela.

Postoje još neke sklopovne[12] arhitekture koje rješavaju isti problem kao LSTM. **Gated recurrent unit** arhitektura je pojednostavljena inačica LSTM arhitekture kojom se postiže brže treniranje modela [7, str. 411-412].



Slika 5: Grafički prikaz LSTM ćelije, preuzet iz [19].

[12]engl. gated

2.4 Enkoder-dekoder arhitekture i mehanizam pažnje

U poglavlju 2.1 smo opisali mreže koje ulazni niz proizvoljne duljine preslikavaju u niz iste duljine i mreže koje niz fiksne duljine preslikavaju u niz proizvoljne duljine. U ovom poglavlju pokazat ćemo kako dizajnirati mrežu koja niz proizvoljne duljine preslikava u niz proizvoljne duljine.

Umjesto treniranja jedne mreže koja radi cijeli posao, ideja je istovremeno trenirati dvije rekurentne mreže. Prva mreža je **enkoder**, koji niz $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T_x)}$ preslikava u vektor $\boldsymbol{c}$, koji zovemo **kontekst**. Druga mreža je **dekoder**, koji na ulazu prima kontekst i generira niz $\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(T_y)}$. Prilikom treniranja, za skup $\{(\boldsymbol{x}_1, \boldsymbol{y_1}), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, potrebno je maksimizirati izraz

$$\frac{1}{N}\sum_{i=1}^{N} \log P(\boldsymbol{y}_i^{(1)}, \ldots, \boldsymbol{y}_i^{(T_{yi})} | \boldsymbol{x}_i^{(1)}, \ldots, \boldsymbol{x}_i^{(T_{xi})}).$$

Primjer 2.2. Želimo konstruirati model koji prevodi rečenice s hrvatskog na engleski jezik koristeći enkoder-dekoder arhitekturu. Enkoder je jednostavan, dovoljno je uzeti standardnu rekurentnu arhitekturu i posljednje skriveno stanje $\boldsymbol{h}^{(T_x)}$ smatrati kontekstom $\boldsymbol{c}$. Za konstrukciju dekodera u vokabular uvodimo riječ koja označava kraj rečenice: `<EOS>`[13]. Za ulazno skriveno stanje u dekoder iskorištavamo kontekst $\boldsymbol{c}$, dok za prvi element ulaznog niza uzimamo riječ `<EOS>`. Za treniranje možemo iskoristiti *teacher forcing* metodu i kao ulaze dekoderu davati točan prijevod $\boldsymbol{y}^{(t)}$, a na kraju zahtijevati da mreža generira `<EOS>`. Prilikom prevođenja neviđenih riječi, umjesto točnog prijevoda $\boldsymbol{y}^{(t)}$ modelu kao ulaz dajemo prethodnu generiranu riječ $\hat{\boldsymbol{y}}^{(t)}$. Dekoder staje kada generira `<EOS>` ili kada generira unaprijed određen maksimalan broj riječi. Prikaz odrolanog grafa računanja za ovakav jezični model se nalazi na slici 6.

Glavni problem kod enkodiranja cijelog ulaznog niza u vektor konteksta je premala dimenzija. Što je ulazni niz duži, teže je opisati sve njegove karakteristike koristeći samo jedan vektor fiksne dimenzije. Bahdanau i suradnici[3], umjesto vektora fiksne duljine za kontekst, u svakom koraku dekodiranja računaju novi kontekstni vektor kojem pridružuju neki podniz ulaznog niza, što nazivaju **mehanizmom pažnje**[14].

Za motivaciju mehanizma pažnje promotrimo problem vizualnog prepoznavanja. Ukoliko na livadi tražimo jabuku, ne možemo dovoljno brzo obraditi svaki dio slike koji nam dolazi do očiju. Kako su livade zelene, a neke jabuke crvene, pažnju ćemo podsvjesno dati svim crvenim predmetima i tako fokusirati vid na dio slike koji vrlo vjerojatno sadrži jabuku.

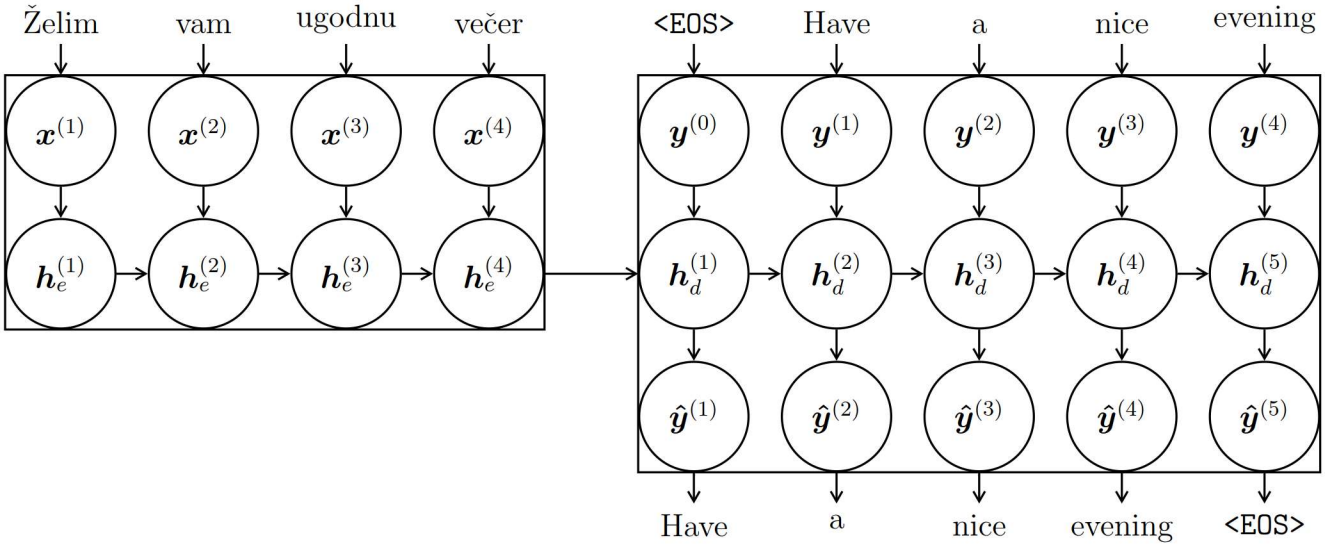
Slično, u obradi prirodnog jezika se za traženu informaciju trebamo fokusirati samo na neki kratak dio teksta, a to nije moguće ako cijeli tekst sažmemo u kontekstni vektor fiksne dimenzije.

2.4.1 Bahdanau pažnja

Kako sada baratamo s dvije rekurentne mreže čiji skriveni slojevi nisu nužno iste dimenzije, za dimenzije skrivenih slojeva enkodera i dekodera redom uvodimo oznake d_e i d_d. Umjesto računanja jednog kontekstnog vektora $\boldsymbol{c}$, za svaki korak dekodiranja $t \in \{1, \ldots, T_y\}$ računamo kontekstni vektor $\boldsymbol{c}^{(t)} \in \mathbb{R}^{d_e}$. S $\boldsymbol{h}_e^{(t)}$ i $\boldsymbol{h}_d^{(t)}$ redom označavamo skrivena stanja enkodera i dekodera. Skriveno stanje dekodera je sada dodatno funkcija kontekstnog vektora.

[13]Standardan token za kraj rečenice u obradi pridodnog jezika, skraćen od *end of sentence*

[14]engl. attention mechanism



Slika 6: Odrolani graf enkoder-dekoder jezičnog modela s oznakama riječi. U slučaju treniranja mreže, gdje želimo točno generirati poznatu rečenicu danu s $\boldsymbol{y}^{(0)}, \ldots, \boldsymbol{y}^{(4)}$. Za računanje skrivenog stanja dekodera $\boldsymbol{h}_d^{(t)}$ koristimo prethodno skriveno stanje i točnu riječ $\boldsymbol{y}^{(t-1)}$, dok generiranu riječ $\hat{\boldsymbol{y}}^{(t)}$ uspoređujemo s točnom $\boldsymbol{y}^{(t)}$, dok zadnju generiranu riječ uspoređujemo s riječi <EOS>. Ukoliko prevodimo nepoznatu rečenicu, nisu poznate točne riječi $\boldsymbol{y}^{(t)}$ pa ne postoje ti vrhovi, nego za računanje skrivenih stanja koristimo prethodnu generiranu riječ $\hat{\boldsymbol{y}}^{(t-1)}$.

Napomenimo da Bahandau i suradnici[3] predlažu korištenje dvosmjernog enkodera te da se dobivena skrivena stanja konkateniraju:

$$\boldsymbol{h}_e^{(t)} = \begin{bmatrix} \overrightarrow{\boldsymbol{h}_e^{(t)}} \\ \overleftarrow{\boldsymbol{h}_e^{(t)}} \end{bmatrix} \in \mathbb{R}^{2d_e},$$

što u praksi često daje bolje rezultate. Vektor $\boldsymbol{c}^{(t)}$ računamo kao težinsku sumu svih skrivenih stanja enkodera:

$$\boldsymbol{c}^{(t)} = \sum_{i=1}^{T_x} \alpha_{ti} \boldsymbol{h}_e^{(i)}, \tag{2.21}$$

odakle dolazi naziv **globalna pažnja**. Težine α_{ti} računamo kao

$$e_{ti} = a(\boldsymbol{h}_d^{(t-1)}, \boldsymbol{h}_e^{(i)}) = \boldsymbol{w}_v^T \tanh\left(\boldsymbol{W}_d \boldsymbol{h}_d^{(t-1)} + \boldsymbol{W}_e \boldsymbol{h}_e^{(i)}\right) \in \mathbb{R} \tag{2.22}$$

$$\alpha_{ti} = \mathrm{softmax}(\boldsymbol{e}_t)_i, \tag{2.23}$$

gdje su $\boldsymbol{w}_v \in \mathbb{R}^v$, $\boldsymbol{W}_d \in \mathbb{R}^{v \times d_d}$ i $\boldsymbol{W}_e \in \mathbb{R}^{v \times 2d_e}$ novi parametri koje je potrebno naučiti.

Naime, vektor $\boldsymbol{h}_e^{(t)}$ sadrži informacije o ulazima $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}$ s naglaskom na $\boldsymbol{x}^{(t)}$ pa ukoliko je težina uz $\boldsymbol{h}_e^{(t)}$ velika, taj ulaz ima velik utjecaj na računanje trenutnog izlaza. Funkcija a iz izraza (2.22) se naziva **aditivna pažnja** te određuje koliko pažnje treba dati vrijednosti $\boldsymbol{h}_e^{(i)}$ prilikom računanja novog skrivenog stanja $\boldsymbol{h}_d^{(t)}$.

Primjer 2.3. Neka je opet zadan problem konstrukcije jezičnog modela za prevođenje teksta te je potrebno rečenicu *Ja te volim* prevesti u *I love you.* Ukoliko koristimo mehanizam pažnje, dekoder će napraviti sljedeće:

1. U prvom koraku će najveću pažnju dati prvom skrivenom stanju, odnosno riječi *ja*.
2. U drugom koraku će najveću pažnju dati trećem skrivenom stanju, odnosno riječi *volim*.
3. U zadnjem koraku će najveću pažnju dati drugom skrivenom stanju, odnosno riječi *te*.

Na slici 7 se nalazi tzv. *heatmap* prijevoda gornje rečenice. Iako se ovaj primjer čini trivijalnim, što i jest, u slučaju dugačkih tekstova attention mehanizam igra puno veću ulogu.

Slika 7: Heatmap koji bi pokazuje kako bi model iz primjera 2.3 dodijelio vjerojatnosti prevedenim riječima. Svjetlija polja odgovaraju većim vjerojatnostima.

2.4.2 Ostali oblici pažnje

Luong i suradnici [10] daju drugi način računanja globalne pažnje, gdje za računanje konteksta umjesto prethodnog skrivenog stanja dekodera $\boldsymbol{h}_d^{(t-1)}$ koriste trenutno stanje $\boldsymbol{h}_d^{(t)}$. Dobiveni kontekst $\boldsymbol{c}^{(t)}$ tada konkateniraju s trenutnim stanjem $\boldsymbol{h}_d^{(t)}$ i koriste ga za daljnje računanje.

Također, u istom radu predstavljaju koncept **lokalne pažnje**, koja rješava problem skupog računanja pažnje nad cijelim, eventualno dugačkim, ulaznim nizom.

Vaswani i suradnici [18] su 2017. godine predstavili tzv. Transformer model, gdje izbacuju rekurentni dio mreže iz *sequence to sequence*[15] modela i koriste sofisticirani mehanizam pažnje.

[15]Naziv za modele koji niz preslikavaju u niz. U ovom radu smo obradili samo enkoder-dekoder arhitekturu, ali ona nikako nije jedini izbor.

3 Klasifikacija programskog koda

Nakon što smo opisali razne oblike unaprijednih i rekurentnih neuronskih mreža, poželjno ih je negdje iskoristiti osim u demonstracijskim primjerima. U ovom poglavlju ćemo pokazati neke metode klasikacije programskog koda i prepoznavanja njegove funkcionalnosti. Neki od razloga analize programskog koda su sljedeći:

- Provjera duplikacije koda, gdje provjeravamo ponavlja li se neki isječak koda više puta u istom projektu.
- Projvera efikasnosti koda. Primjerice, ako je netko ručno implementirao bubble sort algoritam, potrebno je to otkriti i predložiti neki efikasniji algoritam.
- Uklanjanje sigurnosnih propusta, gdje se pokušava izbjeći korištenje opasnih obrazaca koji mogu rezultirati gubitkom ili curenjem osjetljivih podataka.
- Automatsko generiranje koda, gdje na temelju kratkog opisa funkcionalnosti u prirodnom jeziku želimo generirati isječak koda.

Zbog velike količine koda na većim projektima, ručno obavljanje navedenih poslova je zamorno i neefikasno te se javlja potreba za njihovom automatizacijom. Međutim, to nije nimalo lagan posao zbog značajnih razlika između programskih jezika i načina programiranja, što otežava dizajniranje modela koji može analizirati bilo koji programski projekt.

Pretpostavimo da za dani isječak koda trebamo generirati ključne riječi koje opisuju njegovu funkcionalnost. Na slici 8 se nalaze dva semantički jednaka, ali naizgled potpuno različita isječka koji računaju sumu elemenata niza brojeva. Za oba isječka htjeli bismo

```
int sum(int *A, int n)
{
        int ret = 0;
        for (int i = 0; i < n; ++i)
                ret += A[i];
        return ret;
}
```

```
#define N int
#define X (
#define Y )
#define Z 0
#define W =
#define AAA +=
#define R [
#define L ]
N sum X N *A, N n Y
{
        N M W Z;
        for X int I W 0; I < n; ++I Y
        M AAA A R I L;
        return M;
}
```

Slika 8: Gore: Standardna implementacija funkcije koja zbraja elemente polja u jeziku C. Dolje: Pomućena funkcija semantički ekvivalentna gornjoj.

dobiti iste rezultate pa je očito da pristup koji analizira program kao niz znakova neće riješiti zadatak.

U nastavku ćemo dati pregled dva rada koji se bave klasifikacijom i opisivanjem isječaka koda.

3.1 code2seq

Alon i suradnici [2] su 2018. godine definirali model koji na temelju apstraktnog sintaksnog stabla isječka koda određuje:

- Niz ključnih riječi koje opisuju funkciju (code summarization).
- Opis rada funkcije u prirodnom jeziku (code captioning).

Autori koriste različite oznake stanja i ulaza mreže od onih u poglavljima 1 i 2 te ćemo u nastavku koristiti njihove oznake.

3.1.1 Apstraktno sintaksno stablo

Apstraktno sintaksno stablo je struktura podataka koju prevoditelji programskog jezika koriste za jednoznačan prikaz strukture programa te kasnije za semantičku analizu programa. Na listovima stabla, tzv. terminalima, se nalaze nazivi varijabli i konstante. Ostali vrhovi stabla sadrže operatore definirane u programskom jeziku i njihove argumente.

Koristeći AST isječka koda, ideja rada je program promatrati kao niz puteva između svih parova terminala, svaki od tih puteva preslikati u vektor fiksne dimenzije te koristeći enkoder-dekoder arhitekturu s mehaniznom pažnje generirati riječi iz prirodnog jezika.

Prilikom treniranja, uniformno se odabire k puteva u AST-u. Svaki par terminala $(v_1^i, v_{l_i}^i)$ jednoznačno određuje put

$$(v_1^i, v_2^i, \ldots, v_{l_i}^i), \tag{3.1}$$

gdje je l_i duljina puta i.

3.1.2 Opis modela

U dizajnu rekurentnog modela korištena je standardna enkoder-dekoder arhitektura s Luong mehaniznom pažnje iz poglavlja 2.4.2.

Za niz $\{x_1, \ldots, x_k\}$ puteva u AST-u potrebno je generirati vektorsku reprezentaciju z_i svakog puta. Proces se sastoji od nekoliko dijelova:

1. Na svakom vrhu AST-a se nalazi nekakav simbol, primjerice *Integer* ili *Assign*, te je za nepotpuni vokabular od 364 simbola konstruirana matrica ugrađivanja E^{nodes}. Time je postignuto da svaki vrh na temelju simbola dobije svoju vektorsku reprezentaciju. Za enkodiranje puta se koristi dvosmjerni LSTM:

$$\begin{aligned}(h_1, \ldots, h_l) &= \text{LSTM}(E^{nodes}(v_1), \ldots, E^{nodes}(v_l)) \\ \text{path}(v_1, \ldots, v_l) &= [\overrightarrow{h_l}; \overleftarrow{h_1}].\end{aligned}$$

2. Krajevi svakog puta su terminali koji sadrže neke tokene iz koda, primjerice nazive funkcija koji se mogu sastojati od više riječi. Kako bi se ti tokeni preslikali u vektorski prostor, konstruirana je matrica ugrađivanja $E^{\text{subtokens}}$, dok je reprezentacija tokena w zadana s

$$\text{encode_token}(w) = \sum_{s \in \text{split}(w)} E^{\text{subtokens}}(s).$$

3. Vektorske reprezentacije puta i tokena u terminalima se kombiniraju te se dobiva završna reprezentacija puta:

$$z = \tanh\left(W_{in}[\text{encode_path}(v_1 \dots v_l); \text{encode_token}(v_1); \text{encode_token}(v_l)]\right),$$

gdje je $W_{in} \in \mathbb{R}^{(2d_{\text{path}}+2d_{\text{token}})\times \text{d}_{\text{hidden}}}$

4. Početno skriveno stanje dekodera je prosjek reprezentacija svih odabranih putova

$$h_0 = \frac{1}{k}\sum_{i=1}^{k} z_i. \tag{3.2}$$

Uočimo da ulazni niz puteva nije uređen kao u standardnom enkoder-dekoder modelu. Dovoljno je odabrane puteve odvojeno enkodirati i dati ih dekoderu kao u izrazu (3.2).

3.1.3 Treniranje i rezultati

Oba modela su trenirana na jednak način. Vrijednosti parametara su generirane slučajnim odabirom, ovisno o broju težina modela. Za funkciju gubitka je prirodno odabrana funkcije unakrsne entropije, dok je za optimizacijsku metodu odabran stohastički gradijentni spust s Nestorovljevim momentom. Za regularizaciju je uveden dropout na ulazu, s vjerojatnosti 0.25 u prvom i 0.7 u drugom problemu te rekurentni dropout s vjerojatnosti 0.5 u enkoderskom dijelu mreže. Dimenzije ugrađivanja i skrivenih stanja mogu se pronaći u samom radu.

Za model koji treba generirati ključne riječi funkcije autori su odabrali generirati nazive funkcija napisanih u Java programskom jeziku. Modeli su trenirani na tri različita skupa:

- Mali skup, sastavljen od 11 većih projekata. Skup za treniranje se sastoji od 9 projekata, dok se skupovi za validaciju i testiranje sastoje od 1 projekta.
- Srednji skup, sastavljen od 1000 najpopularnijih Github projekata. Skup za treniranje se sastoji od slučajno odabranih 800 projekata, dok skupovi za validaciju i testiranje imaju po 100 projekata.
- Veliki skup, sastavljen od 9550 najpopularnijih Github projekata. Skup za treniranje se sastoji od slučajno odabranih 9000, skup za validaciju od 250 i skup za testiranje od 300 projekata. U ovom skupu ukupno postoji 16 milijuna isječaka koda.

Za određivanje kvalitete modela korištene su metrike *precision*, *recall* i F1:

$$\begin{aligned}
\text{precision} &= \frac{TP}{TP+FP} \\
\text{recall} &= \frac{TP}{TP+FN} \\
\text{F1} &= 2\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}
\end{aligned}$$

gdje je TP broj riječi koje je model točno generirao (true positive), FP broj riječi koje je model krivo generirao (false positive) i FN broj riječi koje je model trebao generirati, ali nije (false negative).

Autori su pokazali kako njihov model daje bolje rezultate po svim evaluacijskim metrikama od ostalih *state-of-the-art* modela. Primjerice, preciznost Transformer modela na

Model	Precision	Recall	F1	ΔF1
code2seq (original model)	**60.67**	**47.41**	**53.23**	
No AST nodes (only tokens)	55.51	43.11	48.53	-4.70
No decoder	47.99	28.96	36.12	-17.11
No token splitting	48.53	34.80	40.53	-12.70
No tokens (only AST nodes)	33.78	21.23	26.07	-27.16
No attention	57.00	41.89	48.29	-4.94
No random (sample k paths in advance)	59.08	44.07	50.49	-2.74

Slika 9: Tablica varijacija rezultata na različitim code2seq modelima. Rezultati su dobiveni na validacijskom skupu srednjeg skupa podataka. Slika preuzeta iz [2].

zadana tri skupa je 5% do 15% manja nego preciznost code2seq modela. Ostali rezultati mogu se pronaći u [2, tablica 1, str. 6]. U usporedbi s ostalim modelima, pokazano je da code2seq nije osjetljiv na male skupove podataka, što je potkrijepljeno usporedbom rezultata na malom skupu podataka od 700000 isječaka koda.

Na kraju je proveden eksperiment u kojemu su dijelovi mreže uklonjeni, a rezultati uspoređeni sa standardnom mrežom. Tablica se može pronaći na slici 9. Zanimljivo je vidjeti da F1 skor padne čak za 12.7% ukoliko se iz ugrađivanja tokena ukloni razdvajanje tokena na više riječi. Uočimo još da se bez mehanizma pažnje F1 skor smanji samo za 4.94%, vjerojatno zbog kratkih isjčaka koda u kojima pažnja nije potrebna.

Za treniranje drugog modela autori su odabrali su CodeNN skup podataka konstruiran od 66015 pitanja i odgovora sa web stranice StackOverflow. Za svaki par postoji isječak koda u C# programskom jeziku i kratak opis funkcionalnosti u prirodnom jeziku, prosječne duljine 10. Za evaluaciju je korišten BLEU[16] skor, koji se općenito koristi za određivanje kvalitete prijevoda teksta. Slično kao u prvom problemu, rezultati modela su bolji od ostalih *state-of-the-art* modela.

3.2 Neural Code Comprehension

Ben-Nun i suradnici [4] su 2018. godine predstavili model koji klasificira isječak koda na temelju njegove funkcionalnosti. Cilj im je bio dati što robusniji model, neovisan o programskom jeziku i nazivima funkcija i varijabli.

Ovisnost o izboru programskog jezika izbjegnuta je korištenjem *LLVM intermediate representation*[17] jezika, u koji se mogu prevesti brojni programki jezici, primjerice C, C++, Python, Rust ili Go. Autori prvo argumentiraju konstrukciju tzv. *grafa konteksnog toka*, koji sadržava različite veze u LLVM IR reprezentaciji programa, te na temelju tih grafova konstruiraju ugrađivanje LLVM IR naredbi u vektorski prostor dimenzije 200, što nazivaju **inst2vec**. Isječak koda se prilikom klasifikacije tako može promotriti kao niz LLVM IR naredbi i klasificirati koristeći rekurentnu neuronsku mrežu. Na slici 10 se vidi skup podataka na kojemu je trenirano inst2vec ugrađivanje.

3.2.1 Opis modela

Za treniranje modela je korišten skup podataka POJ − 104[12], sastavljen od 104 različita algoritma i 500 implementacija svakog algoritma. Za treniranje je slučajno odabrano 60%

[16]Skraćeno od *bilingual evaluation understudy*.

[17]Referenca jezika se može pronaći na `https://llvm.org/docs/LangRef.html`

Discipline	Dataset	Files	LLVM IR Lines	Vocabulary Size	XFG Stmt. Pairs
Machine Learning	Tensorflow [1]	2,492	16,943,893	220,554	260,250,973
High-Performance Computing	AMD APP SDK [9]	123	1,304,669	4,146	45,081,359
	BLAS [22]	300	280,782	566	283,856
Benchmarks	NAS [57]	268	572,521	1,793	1,701,968
	Parboil [59]	151	118,575	2,175	151,916
	PolybenchGPU [27]	40	33,601	577	40,975
	Rodinia [14]	92	103,296	3,861	266,354
	SHOC [21]	112	399,287	3,381	12,096,508
Scientific Computing	COSMO [11]	161	152,127	2,344	2,338,153
Operating Systems	Linux kernel [42]	1,988	2,544,245	136,545	5,271,179
Computer Vision	OpenCV [36]	442	1,908,683	39,920	10,313,451
	NVIDIA samples [17]	60	43,563	2,467	74,915
Synthetic	Synthetic	17,801	26,045,547	113,763	303,054,685
Total (Combined)	—	24,030	50,450,789	8,565	640,926,292

Slika 10: Popis i statistike skupova podataka na kojima je trenirano inst2vec ugrađivanje, preuzet iz [4]

skupa, dok je za validaciju i testiranje odabrano po 20% skupa te je svaki isječak iz skupa na treniranje preveden 8 puta sa različitim zastavicama prilikom prevođenja kako bi se skup povećao. Kao funkcija gubitka je korištena unakrsna entropija te je korišten optimizacijski algoritam Adam.

Model se sastoji od LSTM sloja sa skrivenim stanjima dimenzije 200, batch normalizacije te dva unaprijedna sloja, jedan dimenzije 32 s ReLU aktivacijom, dok je drugi dimenzije 104, što odgovara broju klasa, sa sigmoidnom aktivacijom.

3.2.2 Implementacija i rezultati

U sklopu diplomskog rada dajemo implementaciju mreže u programskom jeziku Python, koristeći popularnu biblioteku za strojno učenje Pytorch[18].

Implementacija se može pronaći na `https://gitlab.com/ajovanov/ncc-reimplementation`. Pošto su autori dali naučeno inst2vec ugrađivanje i razdvojili podatke na skupove za treniranje, validaciju i testiranje, možemo ih iskoristiti za učenje opisanog modela.[19]

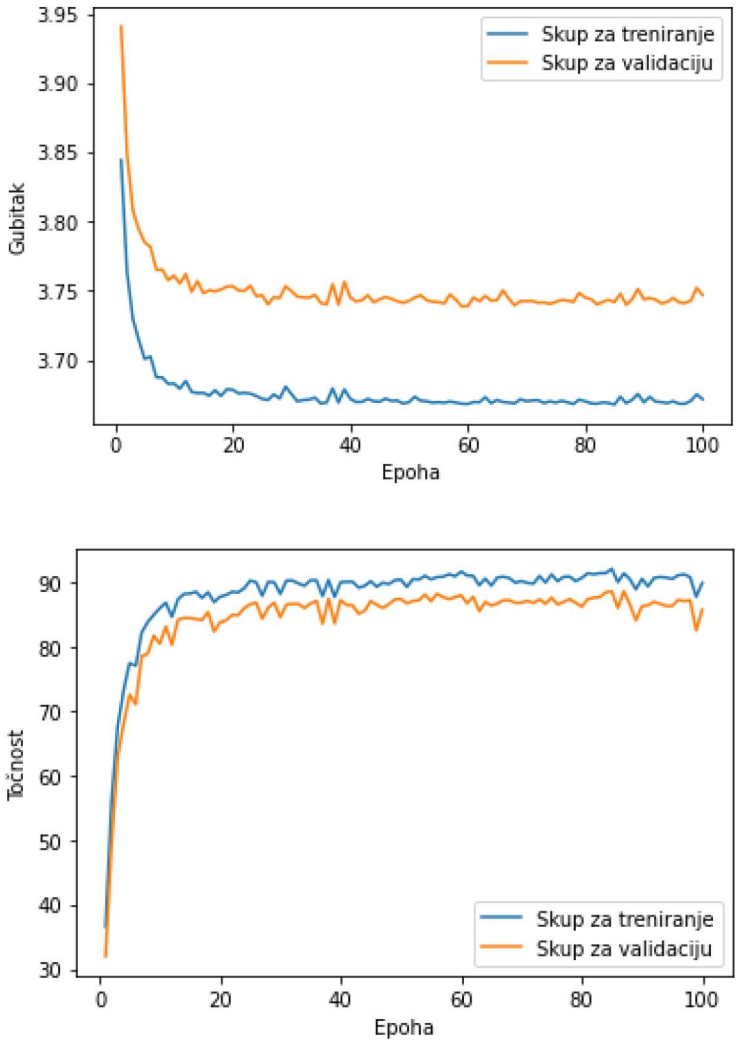
Pytorch je biblioteka razvijena u laboratoriju *Facebook's AI Research lab* te se uglavnom koristi za brzu implementaciju i efikasno treniranje neuronskih mreža. Umjesto ručne implementacije mreža i algoritama učenja, za definiranu mrežu moguće je automatski obaviti propagaciju unaprijed i unazad, izračunati gradijent funkcije gubitka i ažurirati parametre koristeći neku optimizacijsku metodu. Računanje se može znatno ubrzati ukoliko se koristi Nvidia GPU s omogućenim CUDA operacijama.

Točnost modela u radu na testnom skupu iznosi 94.83% nakon 100 epoha treniranja, što je bolje od ostalih *state-of-the-art* metoda klasifikacije. U našoj implementaciji je dobivena točnost 87.98% nakon 100 epoha treniranja, što upućuje na pretreniranost modela jer je točnost na skupu za treniranje veća od 90%. Pošto su modeli implementirani koristeći različite biblioteke, vjerojatan razlog je u različitim postavkama hiperparametara koje je

[18] `https://pytorch.org/`

[19] Ugrađivanje i podaci se mogu pronaći na `https://github.com/spcl/ncc`.

potrebno ručno namjestiti. Također, poželjno je korištenje dodatne regularizacije, primjerice dropout mehanizma. Na slici 11 se nalaze vrijednosti funkcije gubitka i točnosti modela po epohama treniranja.



Slika 11: Gore: vrijednosti funkcije gubitka po epohama na skupovima za treniranje i validaciju. Dolje: točnosti modela na istim skupovima.

Literatura

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

[2] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code, 2019, 1808.01400.

[3] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2016, 1409.0473.

[4] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. Neural code comprehension: A learnable representation of code semantics, 2018, 1806.07336.

[5] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 2011. PMLR.

[6] G. H. Golub and C. F. V. Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MA, SAD, 2013.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT Press, Cambridge, MA, SAD, 2016. `http://www.deeplearningbook.org`.

[8] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9:1735–1780, 11 1997. doi:10.1162/neco.1997.9.8.1735.

[9] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017, 1412.6980.

[10] M. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015, 1508.04025. URL `http://arxiv.org/abs/1508.04025`.

[11] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013, 1310.4546. URL `http://arxiv.org/abs/1310.4546`.

[12] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing, 2015, 1409.5718.

[13] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[14] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1957. doi:10.1037/h0042519.

[15] S. Ruder. An overview of gradient descent optimization algorithms, 2017, 1609.04747.

[16] R. Scitovski, N. Truhar, and Z. Tomljanović. *Metode optimizacije.* Sveučilište J. J. Strossmayera u Osijeku, Odjel za matematiku, Osijek, Hrvatska, 2014.

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2017, 1706.03762.

[19] X. Yuan, L. Li, and Y. Wang. Nonlinear dynamic soft sensor modeling with supervised long short-term memory network. *IEEE Transactions on Industrial Informatics*, PP:1–1, 02 2019. doi:10.1109/TII.2019.2902129.

Sažetak

U ovom radu definiramo unaprijedne i rekurentne neuronske mreže koje se koriste za rješavanje zadataka strojnog učenja. Neuronske mreže su sofisticirani model računanja za koji se može izraditi efikasan algoritam treniranja, procesa kojim neuronska mreža uči kako reprezentirati podatke uzorkovane iz neke nepoznate distribucije.

U prva dva poglavlja rada definiramo razne oblike neuronskih mreža i dajemo primjere problema koje oni rješavaju. Naglasak se stavlja na LSTM arhitekturu, enkoder-dekoder arhitekturu i razne mehanizme pažnje.

U trećem poglavlju motiviramo automatizaciju analize programskog koda. Dajemo pregled radova koji pokazuju kako primijeniti rekurentnu neuronsku mrežu za određivanje kratkog opisa funkcionalnosti isječaka programskog koda te klasifikaciju programskog koda.

Ključne riječi: strojno učenje, neuronske mreže, povratna propagacija, enkoder-dekoder, pažnja, klasifikacija algoritama

Application of recurrent neural networks in code analysis

Summary

In this paper we define feedforward and recurrent neural networks that are used to solve machine learning problems. Neural networks are a sophisticated computational model with an efficient training algorithm, a process where the networks learns how to describe data sampled from an unknown probability distribution.

In the first two chapters, we define different types of neural networks and give examples of problems they can solve. We put emphasis on the LSTM architecture, encoder-decoder architectures and various attention mechanisms.

In the third chapter, we motivate automation of program analysis. We review papers that show how to apply recurrent networks to summarize and caption code snippets and to classify programs.

Keywords: machine learning, neural networks, backpropagation, encoder-decoder, attention, algorithm classification

Životopis

Antonio Jovanović je rođen 1997. godine u Osijeku. Osnovnu školu završava u Višnjevcu, dok III. gimnaziju završava u Osijeku. Obrazovanje nastavlja na preddiplomskom studiju matematike na Odjelu za matematiku Sveučilišta J.J.Strossmayera u Osijeku te ga završava 2019. godine sa završnim radom na temu kompaktnih prostora. Nakon preddiplomskog studija matematike upisuje diplomski studij matematike i računarstva na istom Odjelu.

Tijekom studija je radio kao demonstrator iz kolegija Uvod u vjerojatnost i statistiku. Svake akademske godine između 2016./2017. i 2020./2021. je sudjelovao na IEEEXtreme natjecanju iz programiranja, a akademskih godina 2016./2017. i 2017./2018. na srednjeeuropskom ACM natjecanju iz programiranja CERC.

Dobitnik je rektorove nagrade za akademsku godinu 2018./2019. te pročelnikove nagradu za istu godinu.