UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

1a1e8e70-55ec-4ce4-9091-ffc9d7132b48

PRÍPRAVA ÚLOH NA CVIČENIA

Z REKURENTNÝCH NEURÓNOVÝCH SIETÍ

BRATISLAVA 2011

Matej Pecháč

UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

PRÍPRAVA ÚLOH NA CVIČENIA

Z REKURENTNÝCH NEURÓNOVÝCH SIETÍ

(bakalárska práca)

Študijný program :	aplikovaná informatika
	(Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.9 APLIKOVANÁ INFORMATIKA
Školiace pracovisko:	Katedra aplikovanej informatiky, FMFI UK v Bratislave
Školiteľ:	doc. Ing. Igor Farkaš, PhD.

BRATISLAVA 2011

Matej Pecháč

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Matej Pecháč
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.9. aplikovaná informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Príprava úloh na cvičenia z rekurentných neurónových sietí

Cieľ: Implementujte (v jazyku Python) základné modely rekurentnej siete a učiacich algoritmov RTRL, BPTT a ESN, a vytvorte didaktický produkt užívateľsky prívetivým spôsobom, aby sa dal dobre používať za účelom výuky na cvičeniach z umelých neurónových sietí. Treba využiť existujúci balík PyBrain.

Literatúra: Stránka predmetu Neurónové siete, http://ii.fmph.uniba.sk/farkas/ns.html Schaul T. et al.: PyBrain. Journal of Machine Learning Research 1 (2010) 999-1000.

Poznámka: Požiadavky: programovanie v jazyku Python, absolvovanie predmetu Neural Networks najneskôr v šk. roku 2010/11, angličtina, schopnosť (relatívne) samostatnej práce, pravidelné konzultácie.

Vedúci: doc. Ing. Igor Farkaš, PhD.

Dátum zadania: 01.10.2010

Dátum schválenia: 17.05.2011

doc. RNDr. Mária Markošová, PhD.
garant študijného programu

.....
študent

.....
vedúci

Prehlásenie

Čestne prehlasujem, že som túto prácu vytvoril samostatne, vedený radami a usmerneniami môjho školiteľa, čerpajúc z uvedenej literatúry a zdrojov dostupných na internete.

V Bratislave dňa 3. 6. 2011

.....

Matej Pecháč

Abstrakt

Cieľom tejto bakalárskej práce bolo vytvoriť interaktívnu didaktickú aplikáciu pre študentov študujúcich predmet umelé neurónové siete. Zamerali sa v nej na vybrané modely rekurentných neurónových sietí (Elmanov, Jordanov a ESN). Pripravili sme niekoľko úloh demonštrujúcich možnosti RNS. Implementovali sme aj trénovacie algoritmy BP, BPTT a RTRL. Celá aplikácia pracuje na platforme Unix. Vývoj prebiehal v jazyku python za pomoci knižnice PyBrain.

Goal of this bachelor's thesis was to develop interactive didactic application for students studying artificial neural networks. We focused on chosen models of recurrent neural networks (Elman network, Jordan network and ESN). We prepared a few tasks demonstrating potential of RNN. We also implemented training algorithms BP, BPTT and RTRL. Application runs on Unix platform. It was developed in python with using of PyBrain library.

Obsah

1 Úvod

Cieľom tejto práce je vytvoriť didaktickú aplikáciu pre študentov, určená pre predmet umelé neurónové siete a poskytnúť im čo najinteraktívnejšiu možnosť zapojiť sa do procesu vytvárania, trénovania a testovania rekurentných neurónových sietí (RNS). Zároveň by im mala pomôcť preniknúť k princípom, ktoré za touto problematikou stoja a motivovať ich k ďalšiemu záujmu a vlastnej tvorbe v tejto oblasti.

Preto je dôraz kladený na intuitívne grafické rozhranie a jednoduchosť ovládania. Taktiež aplikácia ponúka široké spektrum nastavení, ktorými sa dájú odskúšať postupy vedúce (a často aj nevedúce) k riešeniu. Výhodou je možnosť jednoduchého vytvárania vlastných trénovacích množín.

Produktom nie je iba samotná aplikácia ale aj sada predpripravených trénovacích množín, ktoré sme starostlivo vybrali, aby pokryli všetky najznámejšie možnosti takýchto sietí. Mali by načrtnúť výpočtovú silu a schopnosť RNS naučiť sa rôzne typy úloh, ktoré sa spomínajú v literatúre a na prednáške predmetu neurónové siete. Študent vďaka tomu nezíska iba teoretickú vedomosť o tom, čo by sa sieť mala vedieť naučiť, ale môže si to priamo overiť a vyskúšať.

Našou snahou je taktiež vyplniť chýbajúci článok v cvičeniach, keďže RNS nemajú v nich zastúpenie. Pre cvičenie ostatných modelov sa používa balík od S. Marslanda vytvorený v pythone. Preto sme sa rozhodli použiť python aj pre našu aplikáciu a dodatočnou motiváciou bola existencia knižnice PyBrain, ktorá implementuje iba dopredné NS. Výsledkom práce by malo byť aj rozšírenie knižnice PyBrain o RNS a algoritmy na ich trénovanie.

Keďže táto téma je možno pre čitateľa novinkou, či nemá úplný prehľad o problematike, v druhej kapitole sa zamieravame na teoretický základ rekurentných neurónových sietí a na stručné vysvetlenie algoritmov, ktoré sú použité v našej práci. V tretej kapitole sa budeme venovať podrobnej špecifikácii projektu, hlavných cieľov, ktoré sme sa snažili dosiahnuť a požiadavkam, ktoré sú naň kladené. Štvrtá kapitola pojednáva o implementácii, kde popíšeme celkový výsledok, netriviálne časti budú zdokumentované pseudokódom. Nasleduje záver, kde zhrnieme výslednú podobu aplikácie a načrtneme možné vylepšenia, ktoré by mohli našu prácu posunúť v budúcnosti ďalej.

2 Analýza rekurentných neurónových sietí a algoritmov ich trénovania

2.1 Stručný prehľad histórie neurónových sietí

V roku 1949 vyslovil kanadský psychológ Donald Hebb hypotézu, že dve nervové bunky, ktoré majú často jedna na druhú excitačný účinok, následne posilňujú svoje spojenia. Z tejto experimentálne overenej predpovede vyplýva, že informácia je distribuovaná prostredníctvom zmeny účinností synaptických spojení.

V roku 1958 Frank Rossenblatt, stavajúc na experimentoch McCullocha a Pittsa, uviedol jednu umelú nervovú bunku, ktorá vedela klasifikovať objekty a nazval ju „perceptrón“. Trénovaná bola jednoduchým pravidlom, ktoré položilo základy pre skupinu algoritmov učenia s učiteľom.

Umelá nervová bunka prebrala všetky rysy svojho biologického podkladu. Mala niekoľko vstupov, ktoré boli prenásobované modifikovateľnými váhami. Následne sa sčítavali v „tele“ umelého neurónu a táto suma ako argument vstupovala do aktivačnej funkcie. Najčastejšie sa používa lineárna, sigmoidálna funkcia alebo hyperbolický tangens. Do aktivačnej funkcie sa často zavádza ešte jeden vstup predstavujúci prahovú hodnotu neurónu. Býva nastavený na +1 alebo -1.

Nevýhodou perceptrónu je jeho obmedzenosť na klasifikáciu lineárne separovateľných problémov, akými sú napr. logické funkcie AND, OR, avšak už nie XOR. V časoch perceptornu eexistovalo žiadne učiace pravidlo, ktoré by dokázalo naučiť sieť, ktorá by tento problém zvládla.

Až v roku 1986 zaviedli Rumelhart, Hinton a Williams nový učiaci algoritmus, ktorý sa udomácnil v slovenskej literatúre ako učenie pomocou spätného šírenia chyby (Backpropagation, BP), ktorý možno aplikovať viacvrstvové siete. Hlavnou myšlienkou sú dva kroky: v prvom sa vstupný signál dopredne šíri sieťou až k výstupu, kde sa vyráta chyba, a následne sa šíri spätne až ku vstupom, pri čom sa podľa nej modifikujú jednotlivé spojenia – váhy. Takýmto algoritmom už bolo možné preklenúť prvotné obmedzenia a bez problémov naučiť sieť funkciu XOR a mnoho iných lineárne neseparovateľných problémov, ktoré dovtedy neboli zvládnuté.

Viacvrstvové siete, ktoré boli učené algoritmom BP, sa skladali zo vstupnej, jednej alebo viacerých skrytých a výstupnej vrstvy. Práve skrytá vrstva má kľúčovú úlohu pre učenie sa problému, pretože na nej dochádza k reprezentácii stavov, či príznakov a je možné ich odtiaľ extrahovať.

Viacvrstvové siete sú schopné po naučení klasifikovať, majú vlastnosť generalizácie objektov, používajú sa ako aproximátory spojitých funkcií.

Zdalo by sa, že takéto siete dokážu zvládnuť naučenie akejkoľvek úlohy a nemajú žiadne obmedzenia. Avšak to nie je pravda. Ukázalo sa, že síce výborne zvládajú zachytávanie priestorovej štruktúry dát, ale to už neplatí o časovej štruktúre.

Môžeme si ľahko predstaviť úlohu, kde nezáleží iba na akomsi „rozmiestnení“ či príznakoch obsiahnutých v predkladanej vzorke, ale aj na časovej postupnosti t.j. na poradí v akom budeme predkladať vzorky sieti. V tomto prípade už viacvrstvové siete ani BP úplne nevystačili (hoci niektoré jednoduchšie úlohy dokázali dostatočne presne zvládnuť) a problémy, ktoré sme práve načrtli volali o novom prístupe v oblasti umelých neurónových sietí.

Riešenie sa objavilo v dvoch verziách, pri čom obe dokázali vyriešiť problém s učením sa dát s časovou štruktúrou.

Prvým prístupom bola sieť s oknom do minulosti (time-delay neural network [8]). Išlo o jednoduché rozšírenie viacvrstvovej neurónovej siete, ktorá mala okrem aktuálneho vstupu aj N vstupov z minulosti a mohla byť trénovaná metódou BP. Nevýhodou v tomto prípade bola nutnosť dopredu odhadnúť dĺžku okna do minulosti a často na tom závisel celkový výsledok. Takáto sieť však zlyháva, ak sa má naučiť reprezentovať napríklad nejaký konečný automat. Vtedy už nie je možné zvoliť nijaké N, ktoré by bolo dostatočné. Výhodnejšie sa preto javilo, ak by bolo možné sieť naučiť interne reprezentovať jednotlivé stavy konečného automatu.

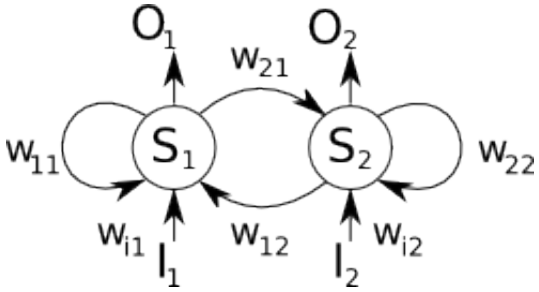
Práve tomuto prístupu vyhovovala nová architektúra sietí, ktoré sa nazývajú rekurentné a podrobne sa im budeme venovať v nasledujúcej podkapitole.

2.2 Plne a čiastočne rekurentné siete

Ako sme už načrtli v minulej podkapitole, budeme sa detailnejšie zaoberať rekurentnými sieťami, keďže tie sú hlavným predmetom tejto práce. Skúsime tu zhrnúť informácie, ktoré by študenti už mali ovládať, a na ktorých sa dá stavať v praxi.

Pod pojmom rekurentná sieť budeme rozumieť sieť, ktorá je rozšírená o vnútornú pamäť v podobe rekurentných spojení. Čiže jej spojenia vytvárajú cyklický graf (v prípade viacvrstvových dopredných sietí to bol acyklický graf). Rekurentné siete možno rozdeliť na plne rekurentné a čiastočne rekurentné.

Plne rekurentné siete obsahujú spojenia neurónov každý s každým, pri čom niektoré neuróny sú vstupné, niektoré skryté a niektoré výstupné (tieto množiny neurónov nemusia byť disjunktné). Okrem toho obsahujú aj rekurentné spojenia samé na seba, čí je zabezpečená pamäť minulých aktivácii. Pre ilustráciu uvádzam obrázok jednoduchej plne rekurentnej siete s dvoma neurónmi.



Obr. 1: Plne rekurentná sieť obsahujúca dva neuróny

Pre aktivácie takýchto neurónov platí jednoduchá rovnica:

$$s_i^{(t+1)} = g\left(\sum_{j=1}^{2} w_{ij} \cdot s_j^{(t)} + i_j\right) \tag{2.1}$$

Aktivácia neurónu v čase t+1 závisí od aktivácii oboch neurónov v čase t a vstupu $\boldsymbol{i}$.

Avšak naša práca sa na tento druh sietí nezameriava, ale pracuje s niekoľkými špeciálnymi prípadmi, ktoré historicky z plne rekurentných sietí vznikli. Išlo o viacvrstvové siete doplnené o tzv. „kontextovú“ vrstvu, ktorá tvorí vstupnú vrstvu v čase t+1, kde sa objavujú aktivácie z času t.

2.2.1 Elmanova sieť

Prvou sieťou, na ktorú sa zamierame je rekurentná sieť, ktorej architektúru navrhol Jeff Elman v roku 1990. Ide o klasickú doprednú dvojvrstvovú sieť, kde kontextová vrstva udržiava kópie neurónov skrytej vrstvy. Sú prenášané identickým spojením s jednotkovou váhou a vstupujú do siete v čase t+1. Kontextová vrstva je prepojená plným spojením so skrytou vrstvou.
Dynamika tejto siete je popísaná nasledujúcimi rovnicami:

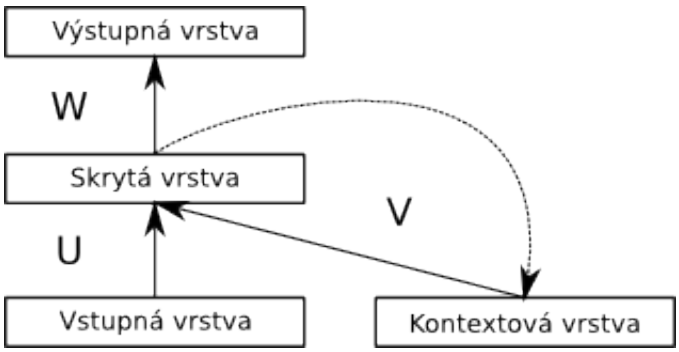
$$\boldsymbol{h}_i^{(t)} = f\left(\sum_{j=1}^{m} u_{ij} \cdot \boldsymbol{i}_j^{(t)} + \sum_{k=1}^{n} v_{ik} \cdot \boldsymbol{h}_k^{(t-1)}\right) \tag{2.2}$$

Aktivácia skrytej vrstvy závisí od vstupu $\boldsymbol{i}$ v čase t a taktiež od predchádzajúcej aktivácií skrytých

neurónov **h** v čase *t*, ktoré sú uložené v kontextovej vrstve. Hodnoty výstupných neurónov potom vypočítame:

$$\boldsymbol{o}_i^{(t)} = f\left(\sum_{k=1}^{n} w_{ik} \cdot \boldsymbol{h}_k^{(t)}\right) \tag{2.3}$$

Môžeme si všimnúť, že takáto sieť má tri matice váh (***W***, ***V***, ***U***), ktoré sú trénovateľné. Na obrázku č. 2 sú naznačené ako plné čiary spájajúce jednotlivé vrstvy. Rekurentné spojenie trénovateľné nie je a na schematickom obrázku je naznačené čiarkovanou čiarou.



Obr. 2: Elmanova sieť

Elamonva sieť bola úspešne použitá na predikciu písmen, či slov vo vetách, ako aj na klasifikáciu slovných druhov.

2.2.2 Jordanova sieť

Je obdobou Elmanovej siete, s tým rozdielom, že kontextová vrstva udržiava kópiu výstupnej vrstvy a tá následne vstupuje do skrytej vrstvy v čase *t*+1. Mechanizmy na prenos kópií sú obdobné ako v Elmanovej sieti.

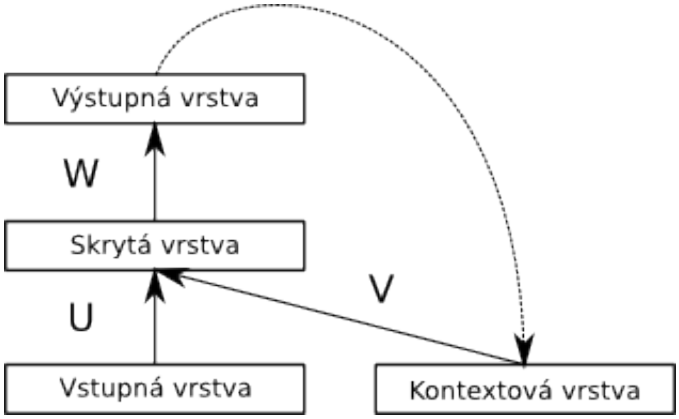
Dynamiku Jordanovej siete popisujú nasledujúce rovnice:

$$\boldsymbol{h}_i^{(t)} = f\left(\sum_{j=1}^{m} u_{ij} \cdot \boldsymbol{i}_j^{(t)} + \sum_{k=1}^{n} v_{ik} \cdot \boldsymbol{o}_k^{(t-1)}\right) \tag{2.4}$$

$$\boldsymbol{o}_i^{(t)} = f\left(\sum_{l=1}^{h} w_{il} \cdot \boldsymbol{h}_l^{(t)}\right) \tag{2.5}$$

Rovnica 2.4 popisuje aktiváciu skrytej vrstvy a rovnica 2.5 popisuje aktiváciu výstupnej vrstvy.

Oproti Elmanovej sieti došlo iba k zámene pri aktiváciách skrytých neurónov, kde nevstupuje do výpočtu skrytá vrstva v čase *t*, za aktivácie výstupnej vrstvy.



Obr. 3: Jordanova sieť

2.3 Formulácia problému

2.3.1 Formalizmus

Ukázali sme už ako sa dajú matematicky popísať dva modely, ktoré budú použité v našej aplikácii, no stále sme formálne nezadefinovali, čo by mal čitateľ rozumieť pod pojmom problém či úloha v kontexte učenia neurónových sietí. Práve tomu sú venované nasledujúce podkapitoly.

Formálne sa dá úloha alebo problém definovať ako funkčný vzťah medzi vstupom $\boldsymbol{i}(t) \in \mathbb{R}^{N_i}$ a požadovaným výstupom $\boldsymbol{d}(t) \in \mathbb{R}^{N_y}$ kde *t* nadobúda hodnoty z intervalu 1…*T*, kde *T* je počet vzorov v dátovej množine obsahujúcej dvojice (***i***(*t*), ***d***(*t*)).

Riešením takto definovaného problému je nájdenie vhodnej funkcie ***y***(*t*) = f(***i***(*t*)), tak, aby chyba *E*(***y***, ***d***) bola minimálna.

$$E(\boldsymbol{y},\boldsymbol{d}) = \frac{1}{2} \cdot \left(\sum_{i=1}^{n} (\boldsymbol{d}_i - \boldsymbol{y}_i)^2 \right) \tag{2.6}$$

2.3.2 Rozšírenie na úlohy bez časovej štruktúry

Existuje niekoľko dobrých matematických nástrojov na riešenie lineárnych rovníc. Avšak

väčšina netriviálnych úloh sa nedá vyjadriť ako jednoduchá sústava lineárnych vzťahov medzi ***i*** a ***d***. Pre mnoho úloh dáva lineárny model $\boldsymbol{y}(t)=\boldsymbol{W}\cdot\boldsymbol{i}(t)$ príliš veľkú chybu pre ľubovoľný operátor ***W***.

($\boldsymbol{W}\in\mathbb{R}^{N_y\times N_i}$)

Riešenie, ktoré sa používa aj v oblasti umelých neurónových sietí je založené na myšlienke rozšírenia vstupu ***i***(*n*) na vysoko rozmerný vektor príznakov:

$$\boldsymbol{x}(t)\in\mathbb{R}^{N_x}$$

V neurónových sieťach predstavujú neuróny skrytej vrstvy (***h)*** práve tento vektor, preto označenie ***x*** zameníme za ***h*** v ďalšom matemaickom popise.

Následne sa nájde vhodná matica, ktorá môže byť vyjadrená nasledujúcim spôsobom:

$$\boldsymbol{y}(t)=\boldsymbol{W}_{\boldsymbol{out}}\cdot\boldsymbol{h}\left(\boldsymbol{i}(t)\right) \tag{2.7}$$

kde platí $\boldsymbol{W}_{\boldsymbol{out}}\in\mathbb{R}^{N_h\times N_y}$, N_h je rozmer skrytej vrstvy, N_i veľkosť vstupného vektoru, N_y je veľkosť výstupnej vrstvy

Typicky platí, že $N_h\gg N_i$. Aj keď neskôr ukážeme, že pre jednoduchšie úlohy to nemusí byť pravidlom.

Zámerne sme z rovníc vynechali aktivačnú funkciu pre zjednodušenie zápisu. Správne by sme predchádzajúcu rovnicu mali prepísať do tvaru:

$$\boldsymbol{y}(t)=f_{out}\left(\boldsymbol{W}_{\boldsymbol{out}}\cdot\boldsymbol{h}\left(\boldsymbol{i}(t)\right)\right) \tag{2.8}$$

kde f_{out} je niektorá nelineárna funkcia používaná ako aktivačná na výstupnej vrstve.

2.3.3 Rozšírenie na úlohy s časovou štruktúrou

Mnoho úloh s časovou štruktúrou možno riešiť na rovnakom princípe. Rozdiel je, že funkcia, ktorú sa má sieť naučiť závisí na histórii predchádzajúcich vstupov (či aktivácii niektorej vrstvy). Preto vektor ***h*** možno prepísať do rekurzívnej formy:

$$\boldsymbol{h}(t)=\left(\boldsymbol{h}(t-1),\boldsymbol{i}(t)\right) \tag{2.9}$$

t je z intervalu 1…*T*, avšak *T* reprezentuje počet krokov diskrétneho času, ktorý potencionálne môže byť aj nekonečný.

Pre rekurentné siete môžeme stavový vektor ***h*** zapísať nasledovne:

$$\boldsymbol{h}(t)=f\left(\boldsymbol{W}_{\boldsymbol{i}}\cdot\boldsymbol{i}(t)+\boldsymbol{W}_{\boldsymbol{h}}\cdot\boldsymbol{h}(t-1)\right) \tag{2.10}$$

Vhodnou substitúciou premenných si môžeme rýchlo všimnúť, že sa jedná o rovnicu skrytej vrstvy popísanú v predošlej podkapitole, kde sme charakterizovali Elmanovu sieť.

$\boldsymbol{W_i}$ a $\boldsymbol{W_h}$ sú matice váhových prepojení medzi vstupnou a skrytou vrstvou a kontextovou a skrytou vrstvou.

2.3.4 Typy úloh s časovou štruktúrou

Úlohy s časopriestorovou úlohou je možné rozdeliť na tri triedy:

- **Klasifikačné alebo asociačné úlohy:**

  Sieť má v tomto prípade rozhodnúť, či práve ukončená časová sekvencia patrí, resp. nepatrí do určitej triedy, prípadne, do ktorej triedy ju možno zaradiť. Najlepším príkladom takejto úlohy je napríklad asociácia vstupov a výstupov automatu s konečným stavom, prípadne rozhodnutie, či slovo patrí, alebo nepatrí do gramatiky nejakého automatu.

- **Predikčné úlohy:**

  Rekurentná sieť by mala nájsť v dátach z intervalu $1 \ldots T$ časovú štruktúru a potom predpovedať nasledujúci vývoj takéhoto radu v čase $t > T$. Takouto úlohou môže byť napríklad predikcia nasledujúcich hodnôt časového radu. Častokrát sa modelujú aj zložitejšie rady vyznačujúce sa chaotickým správaním ako napr. laserové dáta alebo Mackey-Glass časový rad, ktorý vyjadruje dynamiku tvorby bielych krviniek u človeka.

- **Generatívne úlohy:**

  Ide o nadstavbu predikčných úloh, kde sieť nemá iba predikovať údaje na základe „správnych“ vstupov, ale jej predikcie sa následne používajú ako nové vstupy a takto dokáže generovať nové postupnosti. Na začiatku je treba naučenú sieť naštartovať niekoľkými vstupnými signálmi a následne už začne generovať nové.

2.4 Gradientové algoritmy

Po tom, čo sme uviedli modely sietí, ktoré budú použité vo výukovej aplikácii, a ktoré budú mať možnosti študenti učiť rozličné úlohy a taktiež formálne špecifikovali čo znamená pojem úloha, ostáva ešte charakterizovať algoritmy, ktoré slúžia na minimalizáciu chyby a výpočet hľadaných matíc RNS.

Budeme popisovať typ učenia s učiteľom, kedy máme okrem vstupného signálu k dispozícii aj

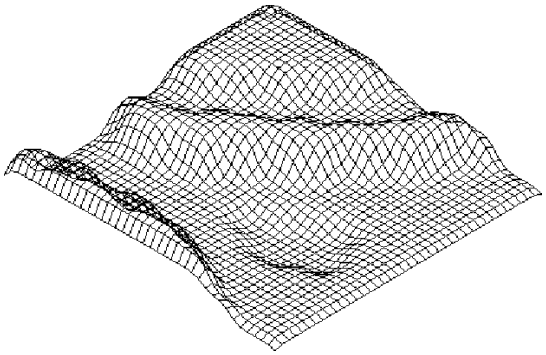
signál učiteľa, na základe ktorého odvodíme chybu siete. Keďže sa jedná o úlohy s časovou štruktúrou, signál učiteľa nemusí byť k dispozícii v každom kroku t, ale napr. iba na konci časovej postupnosti. Tomu sa musia prispôsobiť aj učiace algoritmy.

V našej aplikácii budeme využívať triedu algoritmov používajúcich metódu najprudšieho spádu (steepest descent). Sú založené na myšlienke vypočítania momentálneho gradientu siete na chybovom funkcionáli a následnej adaptácii (úprave) váh proti smeru tohto gradientu. Na tomto mieste je nutné podotknúť, že nikdy nedosiahneme chybu 0 (berieme v úvahu výpočet chyby podľa rovnice 2.11). Tá bude predstavovať asymptotu, ku ktorej sa sieť bude môcť iba priblížiť.

Chybový funkcionál E v čase t (z dôvodu, že všetky úlohy, ktoré budú pre výukovú aplikáciu vytvorené, majú časovú štruktúru, budeme uvažovať stále už iba o trénovacích dátach s časovou štruktúrou) môžeme matematicky zapísať nasledovne:

$$E(t) = \frac{1}{2} \cdot \sum_{i=1}^{n} \left(\boldsymbol{d}_i(t) - \boldsymbol{y}_i(t) \right)^2 \tag{2.11}$$

Pre lepšiu predstavivosť si môžeme chybový funkcionál predstaviť ako reliéf krajiny, v ktorej sa nachádzame na nejakom bode. Jediné čo vieme, je, ktorým smerom sa dá klesnúť. Našou úlohou je klesnúť čo najnižšie t.j. nájsť čo najmenšie lokálne minimum.



Obr. 4: Ilustračný obrázok chybového funkcionálu

Keďže naša pomyselná krajina môže mať aj nejaké lokálne údolia (minimá na funkcionáli), je dobré vyvinúť vhodné mechanizmy, ktoré by zabránili, aby sme v takomto lokálnom minime uviazli. Preto sa popri pevných algoritmoch používajú aj rôzne heuristické vylepšenia, ktoré by spomínanému uviaznutiu mali zabrániť.

Skôr ako bližšie popíšeme spomínané heuristiky, ešte je nutné spomenúť niekoľko spoločných pojmov, ktoré sa vyskytujú v opise každého algoritmu.

- **Rýchlosť učenia:**

  Je veličina, ktorá charakterizuje rýchlosť adaptácie váh. Hovorí aká časť zmeny váh $\Delta \boldsymbol{W}$ sa pripočíta k existujúcim váham. Pre rôzne algoritmy, či typy úloh sa mení, avšak je z pevného intervalu (0,1> a kvantifikuje mieru zmeny váh. Explicitne nie je zakázané použiť aj vyššie rýchlosti učenia ako 1, no častokrát to môže viesť k divergencii váh a nenaučeniu siete.

- **Epocha:**

  Týmto pojmom budeme označovať úsek, kedy algoritmus prejde jedenkrát celú trénovaciu množinu. Najbližšie tri algoritmy, popísané v nasledujúcich podkapitolách, potrebujú na nájdenie globálneho minima viac ako jednu epochu. Obvykle sa táto hodnota pohybuje v rádovo stovkách až tisícoch. Záleží to samozrejme od náročnosti úlohy. Na konci kapitoly si priblížime iný typ siete a algoritmus na jej trénovanie, ktorému stačí jedna epocha na výpočet váhovej matice $\boldsymbol{W}$ (pretože je možný analytický výpočet optimálnych váh). Tento typ siete bude v aplikácii dopĺňať už spomenutú Elmanovu a Jordanovu sieť.

- **Sekvenčné a dávkové učenie:**

  Keďže máme trénovacie množiny s veľkým počtom vzoriek, ponúka sa možnosť adaptovať váhy dvoma spôsobmi. Tým prvým je sekvenčné učenie, kedy váhy zmeníme ihneď ako algoritmus vypočíta ich zmenu. Vtedy sledujeme trajektóriu vedúcu ku globálnemu minimu po dlhšej dráhe a môže to predĺžiť trvanie učenia. Naproti tomu máme dávkové učenie, kedy robíme adaptáciu váh až po prejdení všetkých vzoriek a naraz na konci epochy zmeníme váhy. Sieť tým môže lepšie sledovať trajektóriu a rýchlejšie sa naučiť daný problém.

Teraz po vysvetlení spoločných pojmov pre všetky algoritmy sa môžeme dostať k popisu heuristík, ktoré pomáhajú udržať sieť v dynamickom režime a zároveň jej môžu zabrániť v uviaznutí, v niektorom z lokálnych miním, čo je pre učenie veľmi nežiaduce.

- **Momentum:**

  je veličina, ktorá sa pridáva do rovnice zmeny váh a vyjadruje akúsi zotrvačnosti v učení siete. Jednotkou sú percentá a určuje, koľko percent zmeny váh z predchádzajúceho kroku má byť prirátaných k aktuálnej zmene váh.

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) + \Delta \boldsymbol{W}(t) + \mu \cdot \Delta \boldsymbol{W}(t-1) \tag{2.12}$$

  kde premenná $\mu \in (0,1)$ vyjadruje mieru zotrvačnosti (momentum).

Použitie momenta môže často sieť posunúť z lokálneho minima von a dokáže nájsť nový spád do globálneho minima. Pre niektoré úlohy, ktoré budú priložené k našej aplikácii má momentum zásadný význam a je vhodné ho použiť.

- **Pokles váh:**

  je technika, ktorá by mala odstrániť nepotrebné váhy. Jej podstatou je pri každej adaptácii znížiť adaptované váhy o desatinu, či stotinu percenta.

$$\boldsymbol{W}(t+1)=\boldsymbol{W}(t+1)\cdot\eta \tag{2.13}$$

  kde práve η predstavuje mieru rozpadu váh.

- **Malé iniciálne váhy:**

  Cieľom je inicializovať sieť do dynamického režimu. Neuróny siete najčastejšie používajú sigmoidálnu aktivačnú funkciu, ktorá je „aktívna“ pri vstupnom intervale (-0.5;+0.5). Pre vyššie/nižšie hodnoty začnú byť výstupy blízke 1 resp. 0. Podobne to platí aj pre hyperbolický tangens.

- **Normalizácia vstupu:**

  umožňuje použitie sigmoidy alebo tangnetoidy pre dáta, ktoré majú obor hodnôt mimo oborov hodnôt týchto funkcií.

2.5 Backpropagation (BP)

Bol uverejnený v roku 1986 Rumelhartom, Hintonom a Williamsom [1] a je primárne určený na trénovanie sietí s dopredným šírením. Avšak našiel svoje uplatnenie aj pri trénovaní rekurentných sietí. Princípom algoritmu je výpočet zmeny váh podľa parciálnej derivácie chybového funkcionálu *E* podľa jednotlivých váh:

$$\Delta w=-\alpha\cdot\frac{\partial E}{\partial w} \tag{2.14}$$

Následnou úpravou tohto vzťahu by sme dostali dve nasledujúce rovnice:
pre váhy medzi skrytou a výstupnou vrstvou:

$$\Delta w_{ik}=\alpha\cdot\boldsymbol{\delta_i}\boldsymbol{h_k} \tag{2.15}$$

kde $\boldsymbol{\delta_i}=(\boldsymbol{d_i}-\boldsymbol{y_i})\cdot f_i'$ je vlastne chybou výstupných neurónov a f_i' je derivácia aktivačnej funkcie výstupnej vrstvy.
Pre váhy medzi nižšou skrytou vrstvou prípadne vstupnou/kontextovou a vyššou skrytou vrstvou:

$$\Delta v_{kj} = \alpha \cdot \delta_k \boldsymbol{x}_j \tag{2.16}$$

kde $\boldsymbol{\delta_k} = \left(\sum_i w_{ik} \cdot \boldsymbol{\delta_i} \right) \cdot f'_k$ a f'_k je derivácia aktivačnej funkcie skrytej vrstvy

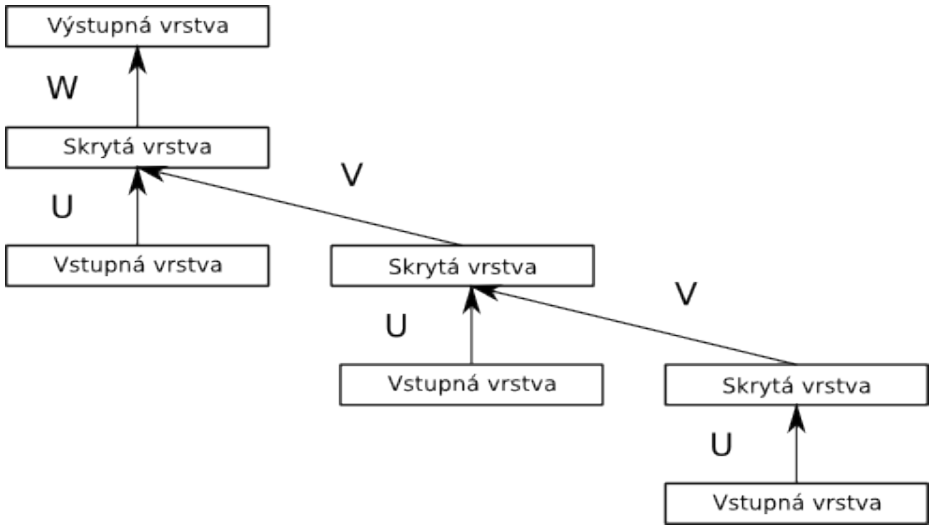
Algoritmus sa dá stručne popísať v týchto krokoch:

- aktivácia siete a výpočet výstupných hodnôt $\boldsymbol{y}$
- výpočet chyby podľa vyššie uvedenej rovnice
- spätné šírenie chyby – výpočet hodnôt $\boldsymbol{\delta_i}$ a $\boldsymbol{\delta_k}$
- adaptácia váhových matíc $\boldsymbol{U}$, $\boldsymbol{V}$ a $\boldsymbol{W}$
- opakovanie celého postupu, prípadné zastavenie, ak je chyba už dosť nízka

S algoritmom môžu byť použité všetky spomínané heuristické vylepšenia. Je možné ho použiť iba na úlohy, ktoré majú k dispozícii signál učiteľa v každom kroku *t*.

2.6 Algoritmus BPTT

Algoritmus BPTT (backpropagation through time) je rozšírením základného algoritmu BP, publikované P. Werbosom v roku 1990 [2]. Problém s časovou závislosťou dát rieši rozvinutím rekurentnej siete podľa dĺžky vstupného signálu (ozn. *T*) na sieť s dopredným šírením. Pre lepšiu predstavu pripájame ilustračný obrázok.



Obr. 5: Ilustračný obrázok rozvinutej Elmanovej siete v dvoch krokoch diskrétneho času

Na takto rozvinutú sieť je možné následne aplikovať obyčajný algoritmus BP. Takto vo výpočte vystupujú jednotlivé kópie váh v a u. Zmenu váh potom možno definovať nasledujúcou rovnicou:

$$\Delta w_{ij} = -\alpha \cdot \frac{\partial E}{\partial w_{ij}} = -\alpha \cdot \sum_t \frac{\partial E}{\partial w_{ij}(t)} = \alpha \cdot \sum_t \boldsymbol{\delta_i}(t) \cdot \boldsymbol{x_j}(t-1) \tag{2.17}$$

kde pre $t = T$:

$$\boldsymbol{\delta_i} = f_i' \cdot \boldsymbol{e_i}(t) \tag{2.18}$$

$\boldsymbol{e_i(t)}$ je chyba vypočítaná vzťahom $\boldsymbol{e_i}(t) = \boldsymbol{d_i}(t) - \boldsymbol{y_i}(t)$

a pre $1 < t < T$:

$$\delta_i = f_i' \cdot \left(e_i(t) + \sum_l w_{il} \cdot \delta_t(t+1) \right) \tag{2.19}$$

l ide cez všetky neuróny skrytej vrstvy.

Stručne môžeme jeho fungovanie zhrnúť do niekoľkých krokov:

- rozvinutie siete na dĺžku T
- aktivácia siete vstupným signálom $\boldsymbol{x}(1)$ až $\boldsymbol{x}(T)$
- výpočet chyby $\boldsymbol{e}(T)$
- výpočet $\boldsymbol{\delta_i}(t)$ pre $t = T \ldots 2$
- adaptácia váhových matíc $\boldsymbol{W}$, $\boldsymbol{V}$, $\boldsymbol{U}$
- opakovanie celého procesu pre ďalšiu postupnosť, prípadné zastavenie

Takýto algoritmus má pamäťovú zložitosť O($N.T$) kde N je počet neurónov a T počet krokov diskrétneho času. Výpočtová zložitosť je O(N^2). Nevýhodou sú veľké pamäťové nároky pre dlhé postupnosti (veľké T). Hoci sa metóda BPTT neujala ako široko používaná trénovacia metóda, úspešne ňou bol simulovaný posuvný register a je aplikovateľná na klasifikačné úlohy [5]. Je určený na špecifickú triedu úloh, ktoré majú definovaný signál učiteľa až na konci celej postupnosti.

2.7 Algoritmus RTRL

Algoritmus RTRL (real-time recurrent learning) bol publikovaný v článku Williamsa a Zipsera v roku 1989 [3]. Jedná sa o posledný nami spomínaný algoritmus najprudšieho spádu. Oproti BPTT má výhodu, že adaptuje sieť online a nepotrebuje preto sieť nijako rozvíjať v čase, aby zachytil históriu podávaných dát.

Neuróny siete rozdeľuje na vstupné a stavové (obvykle skrytá a výstupná vrstva). Ponúka možnosť, aby iba niektoré stavové neuróny mali iba v niektorom čase definovaný signál učiteľa. Tým získava širokú možnosť použitia na všetky typy úloh.

Chybu možno vyjadriť nasledujúcou rovnicou:

$$\boldsymbol{e}_{\boldsymbol{i}}(t) = \boldsymbol{d}_{\boldsymbol{k}}(t) - \boldsymbol{y}_{\boldsymbol{k}}(t) \tag{2.20}$$

ak $k \in T(t)$

$T(t)$ vyjadruje, či má k-tý neurón v čase t učiaci signál alebo nie.

$$\boldsymbol{e}_{\boldsymbol{i}}(t) = 0 \tag{2.21}$$

ak $k \notin T(t)$

Následne chybový funkcionál vyjadríme rovnicou:

$$E(t) = \frac{1}{2} \cdot \sum_{k} \left(\boldsymbol{e}_{\boldsymbol{k}}(t) \right)^2 \tag{2.22}$$

k ide cez všetky stavové neuróny.

Ak chceme robiť online úpravy, musíme vypočítať príspevok každej váhy na výstup každého stavového neurónu, čiže potrebujeme algoritmus, ktorý by rátal túto rovnicu:

$$\Delta w_{ij}(t) = -\alpha \cdot \frac{\partial E(t)}{\partial w_{ij}} = -\alpha \cdot \sum_{k} \frac{\partial E(t)}{\partial \boldsymbol{s}_{\boldsymbol{k}}(t)} \cdot \frac{\partial \boldsymbol{s}_{\boldsymbol{k}}(t)}{\partial w_{ij}} \tag{2.23}$$

ktorú môžeme následne upraviť do konečnej podoby v akej ju budeme používať:

$$\Delta w_{ij}(t) = \alpha \cdot \sum_{k} \boldsymbol{e}_{\boldsymbol{k}}(t) \cdot f'_{k}\left(\boldsymbol{s}_{\boldsymbol{k}}(t)\right) \left[\sum_{l} w_{kl} \frac{\partial \boldsymbol{z}_{\boldsymbol{l}}(t-1)}{\partial w_{ij}} + \delta_{ik} \boldsymbol{z}_{\boldsymbol{j}}(t-1) \right] \tag{2.24}$$

kde je nutné vysvetliť niektoré premenné.

$\boldsymbol{e}_{\boldsymbol{k}}(t)$ je chyba, ktorú sme už zaviedli,

$\boldsymbol{s}_{\boldsymbol{k}}(t)$ je aktivácia stavového neurónu v čase t,

$\boldsymbol{z}_{\boldsymbol{j}}(t)$ je aktivácia j-teho neurónu celej siete v čase t t.j. zjednotenie vstupných a stavových

neurónov,

δ_{ik} je Kroneckerova delta, ktorá vracia 1 ak $i = k$ a 0 v inom prípade

Špeciálne sa zastavíme pri člene $\frac{\partial \boldsymbol{s_k}(t)}{\partial w_{ij}}$ ktorý je mierou citlivosti výstupnej hodnoty $\boldsymbol{s_k}(t)$ na zmenu váhy w_{ij}. Tento člen vstupuje rekurzívne do výpočtu a preto je vhodné položiť iniciálnu rovnicu:

$$\frac{\partial \boldsymbol{s_k}(t_0)}{\partial w_{ij}} = 0$$

Ako sme už spomenuli, algoritmus RTRL má široké použitie na každý typ zo spomínaných úloh. Jeho jedinou nevýhodou je výmena pamäťovej nenáročnosti (keďže adaptácia váh prebieha v každom kroku t) za vyššiu výpočtovú zložitosť $O(N^4)$ oproti dobrej pamäťovej zložitosti $O(N^3)$. Napriek tomu sa dajú nájsť možnosti ako algoritmus sparalelizovať, tak aby sa úprava každej váhy počítala na jednom procesore (napr. takúto možnosť má zapojenie grafických kariet do výpočtu, čo samozrejme vyžaduje nadpriemerné programátorské schopnosti). Vtedy výpočtová zložitosť klesne na $O(N^2 . \log(N))$.

2.8 Sieť s echo stavmi a jej trénovanie

Keďže sme sa rozhodli písať túto kapitolu chronologicky podľa vývoja akým sa neurónové siete uberali, ostáva ešte spomenúť jeden z najnovších modelov, ktorý sa objavil v roku 2001 a jeho autorom je H. Jaeger [6,7].

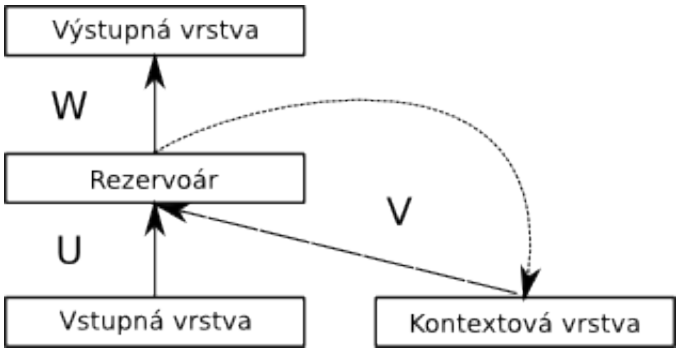
Ide o odlišný prístup k výpočtu matice váh a riešeniu problémov (aj keď uvidíme, že nie je až taký výrazný). Ako už napovedá nadpis, tento model nesie názov sieť s echo stavmi (echo state network, ESN). Obsahuje skrytú vrstvu a výstupnú vrstvu. Môže, ale nemusí, obsahovať vstupnú vrstvu. Spojenia môžu viesť aj z výstupnej vrstvy späť do skrytej vrstvy. Matica váh prepájajúcich kontextovú a skrytú vrstvu (ktorú budeme ďalej nazývať rezervoár) má nasledujúce vlastnosti:

- Je značné veľká, z čoho vyplýva, že počet neurónov rezervoára je v rádovo desiatkach až tisíckach.
- Je veľmi riedka, iba 20% zo všetkých možných spojení sa realizuje, t.j. ich váhy majú nenulové hodnoty, najčastejšie sú inicializované symetricky okolo hodnoty 0 napr. z intervalu <-1,+1>.
- Spektrálny rádius matice (najväčšie vlastné číslo) je v intervale (0,1). Určuje akúsi hĺbku pamäte do minulosti. Nastavenie spektrálneho rádiusu sa mení pre jednotlivé úlohy. Siete

riešiace problémy s potrebou dlhšej pamäte by mali mať spektrálny rádius bližší k 1, zatiaľ čo potreba kratšej pamäte vyžaduje rádius blížiaci sa k 0. Sieť však dlhšou pamäťou môže strácať „zmysel“ pre detail a mať tendenciu príliš generalizovať, zatiaľ čo opačný prípad vedie k zachyteniu podrobností avšak horšiemu modelovaniu dlhých časových závislostí.

V takto vytvorenom rezervoári je teda možné udržiavať „ozveny“ z historických aktivácií a odtiaľ vzniklo aj pomenovanie tejto architektúry.

Príklad takejto siete môžeme vidieť na obrázku č. 6, kde hrubá bodkovaná čiara predstavuje riedke spojenie a husto bodkovaná čiara predstavuje rekurentné spojenie, ktoré skopíruje aktiváciu rezervoára v čase *t* a uloží ju do kontextovej vrstvy.



Obr. 6: Sieť s echo stavmi

Dynamiku takéhoto modelu siete možno popísať nasledujúcimi rovnicami:

$$\boldsymbol{r}_{\boldsymbol{i}}^{(t+1)} = f\left(\sum_{j} u_{ij} \cdot \boldsymbol{i}_{\boldsymbol{j}}^{(t)} + \sum_{k} v_{ik} \cdot \boldsymbol{r}_{\boldsymbol{k}}^{(t)}\right) \tag{2.25}$$

$$\boldsymbol{o}_{\boldsymbol{i}}^{(t+1)} = f\left(\sum_{k} w_{ik} \cdot \boldsymbol{r}_{\boldsymbol{k}}^{(t+1)}\right) \tag{2.26}$$

Kde rovnica 2.25 popisuje aktivácie rezervoára a rovnica 2.26 aktivácie výstupnej vrstvy:

Takto skonštruovaná sieť sa dokáže rýchlo a účinne naučiť hociktorý, zo spomenutých typov úloh. V našej práci sa budeme sústrediť hlavne na predikčné úlohy, keďže primárne na ňu budú zamerané.

Určite treba spomenúť, že bola vyvinutá celá rada techník, ktoré sa trénovaním ESN zaoberali. Okrem metódy, ktorá dokáže upraviť všetky váhy v jednom kroku, tak aby už bola sieť adaptovaná na daný problém, existujú aj rôzne iteratívne metódy založené na výpočte chyby a následných úpravách

matíc, podobné už spomínaným algoritmom. My sa však sústredíme na teoretický opis metódy jednokrokového výpočtu váh, lebo práve tá bude použitá v aplikácii.

Na úvod si musíme stanoviť iniciálne podmienky, ktoré sa budú odlišovať od ostatných spomenutých prípadov. Totižto, jediné váhy, ktoré sa trénujú sú váhy spájajúce rezervoár s výstupnou vrstvou. Ostatné ostávajú na začiatku náhodne inicializované a fixované. Matica rezervoárového spojenia sa riadi princípmi, ktoré boli spomenuté v predošlej podkapitole t.j. musí byť riedka, dostatočne veľká a spektrálny rádius musí mať vhodnú hodnotu. Ohľadom inicializácie rezervoára bolo vyvinutých viacero techník, ktoré mali zvýšiť presnosť ESN, no v tejto práci sa nimi nebudeme zaoberať. Všetkým môžeme odporučiť pozrieť článok Jaegera a Lukoševičiusa [7], kde nájdu podrobné informácie aj o tejto téme.

Samotný algoritmus pozostáva z dvoch hlavných krokov:

- Ukladanie aktivácií

  Sieť aktivujeme na nejakej časti trénovacej množiny (napr. prvých 25%) aby sme potlačili úvodný stav vytvorený náhodnou inicializáciou siete. Po prejdení tohto úseku si už môžeme byť istí, že výstupy siete sú skutočnými odozvami na vstupný signál, a že počiatočný stav už vymizol.

  Pri ďalších 75% vstupov si ukladáme do matice $\boldsymbol{M}$ aktivácie rezervoára pre každý vstup (t.j. pre každý časový okamih t) čím dostaneme maticu $\boldsymbol{M}$ s rozmermi R x N, kde R je veľkosť rezervoára a N počet vstupov. Rovnako skonštruujeme maticu $\boldsymbol{T}$, do ktorej budeme ukladať signál učiteľa pre každý časový okamih t. Jej rozmery budú N x O, kde N je počet očakávaných výstupov (zhodný s počtom vstupov) a O je počet neurónov výstupnej vrstvy. Ak máme iba jeden výstupný neurón, $\boldsymbol{T}$ bude stĺpcový vektor Nx1.

- Výpočet výstupných váh

  Teraz môžeme vypočítať výstupné váhy tak, že signál učiteľa $\boldsymbol{d}(t)$ je aproximovaný ako lineárna kombinácia vnútorných aktivácií rezervoára $\boldsymbol{r}_i(t)$. Tento postup je možné matematicky zapísať rovnicou:

  $$\boldsymbol{d}(t) \approx \boldsymbol{y}(t) = \sum_i w_i \boldsymbol{r}_i(t) \tag{2.27}$$

  Z matematického hľadiska ide o vypočítanie lineárnej regresie váh w_i pre regresiu $\boldsymbol{d}(t)$ na stavoch siete $\boldsymbol{r}_i(t)$. Úloha teda spočíva vo výpočte pseudoinverznej matice k matici $\boldsymbol{M}$ a následnom vynásobení maticou $\boldsymbol{T}$.

  $$\boldsymbol{W} = \boldsymbol{M}^{+} \,.\, \boldsymbol{T} \tag{2.28}$$

Tento postup netreba opakovať, keďže maticu $\boldsymbol{W}$ získame hneď v prvom kroku algoritmu uvedeným výpočtom. Sieť, ktorá bude naučená týmto algoritmom dosahuje veľmi dobré výsledky.

3 Špecifikácia požiadaviek pre aplikáciu

3.1 Cieľ práce a cieľová skupina

Hlavným cieľom našej práce je vytvorenie prostredia, v ktorom si budú môcť študenti zaoberajúci sa problematikou rekurentných sietí, ľahko vytvoriť požadovanú sieť, pri čom budú mať k dispozícii tri rôzne architektúry.

Následne si budú môcť zvoliť, niektorú z predpripravených úloh, ktoré budú pokrývať čo najväčšiu škálu možných problémov riešených pomocou RNS. Na tejto dátovej množine budú môcť sieť trénovať pomocou algoritmov, ktoré budú k dispozícii.

Dôraz bude kladený na jednoduché a intuitívne ovládanie, aby študentov neodrádzala svojou zložitosťou, či ťažkopádnosťou. Ďalej sa budeme snažiť študentom ponúknuť čo najširšiu škálu nastavení, možností, ktoré si budú môcť zapnúť, vypnúť, pridať či ináč nastaviť, aby mohli odskúšať hocijaký postup, ktorý ich napadne, prípadne bude odporúčaný.

Dôležitou vecou, ktorú sa budeme snažiť pokryť je čo najväčšia všeobecnosť pri implementácii, aby bola aplikácia čo najmenej obmedzená a mohla sa používať pre veľké množstvo rozdielnych úloh. Samozrejme sa nebude dať postihnúť každá možnosť, či potreba, avšak mala by postačovať na ilustráciu preberanej látky a jej „zhmotnenie“.

Jednou z náročných veci, ktoré bude treba vyriešiť bude vizualizácia výsledkov, chyby či iných parametrov. Len veľmi málo študentov, ktorí sa začali zaoberať touto problematikou, napovedia rady desatinných čísiel, a preto sa budeme snažiť, aby na obrazovkách uvideli všetko, čo sa zobraziť dá a tak snáď mali lepšiu predstavu o procesoch učenia, testovania, či vedeli si predstaviť architektúru, ktorú práve vytvorili.

Neposlednou vecou bude aj snaha implementovať jednoduchý systém načítavania dátových množín, tak aby si šikovnejší študent (alebo cvičiaci) dokázal vytvoriť vlastnú množinu a nebol iba odkázaný na úlohy, ktoré budú predpripravené. Preto sa posnažíme so samotnou aplikáciou distribuovať manuál, kde sa dočíta podrobnosti o vytváraní vlastnej dátovej množiny a zároveň aj malých utilít, ktoré budú použité na vytvorenie cvičení. Ich malou modifikáciou bude možné vytvoriť vlastné dáta taktiež.

Ako už bolo niekoľkokrát spomenuté, cieľovou skupinou našej práce sú študenti. Preto je čo **najpotrebnejšie** vyjsť ich **potrebám**. Naša aplikácia nemá za cieľ naučiť ich konkrétne implementácie

algoritmov. To sa bez ich samostatnej práce a záujmu nedá a sú na to určené semestrálne úlohy. Hlavne by sa mali oboznámiť s chovaním sietí, s možnými priebehmi chybových funkcií a mali by si vybudovať intuíciu, kedy použiť správny postup na dosiahnutie žiadaného výsledku.

Mnohokrát sa v tejto oblasti stalo aj nám, že sme strávili niekoľko hodín, či dokonca dní hľadaním chyby v implementácii algoritmu, lebo sa javilo, že sieť sa nevie daný problém vôbec naučiť. Nakoniec sa ukázalo, že nám len chýbala trpezlivosť a bolo potrebné predĺžiť učenie, či bolo potrebné vhodné nastavenie učiacich parametrov. Preto ak sa študent oboznámi s predpripravenými úlohami a sám ich všetky odskúša a úspešne zvládne ich natrénovanie, malo by mu to pomôcť pri jeho vlastných implementáciách a mal by predísť márnemu hľadaniu chyby a nabrať odvahu na experimentovanie.

3.2 Opis funkčnosti

3.2.1 Grafické rozhranie

Užívateľ bude mať k dispozícii grafické rozhranie, z ktorého bude mať prístupné všetky ovládacie prvky. Grafické rozhranie bude interagovať s užívateľovými akciami a sprístupňovať resp. znemožňovať mu prístup k ďalšiemu ovládaniu na základe jeho akcií. Keďže niektoré akcie majú svoju logickú súslednosť a nie je možné zameniť ich poradie, grafické rozhranie bude vždy ponúkať iba akcie, ktoré budú mať korektný dopad na celkové dianie.

3.2.2 Vytvorenie siete

Bude obsluhované jednoduchým dialógovým oknom, v ktorom bude môcť užívateľ zvoliť architektúru siete z troch možností:

- Elmanovu sieť
- Jordanovu sieť
- Sieť s echo stavmi

Následne bude môcť zvoliť veľkosti jednotlivých vrstiev, pripojiť prahovú jednotku. V prípade siete s echo stavmi aj nastaviť spektrálny rádius.

Ak užívateľ zadá nulové hodnoty pre niektorú z veľkostí vrstiev, vytvorenie by nemalo prebehnúť a v konzole by sa mal užívateľ dozvedieť chybu, keďže všeobecnosť nášho riešenia sa obmedzuje na spomínané tri modely sietí. Vytvorenie siete odomkne možnosti načítania dátovej množiny.

3.2.3 Načítanie dátovej množiny

Kvôli čo najväčšej flexibilite aplikácie budeme riešiť dátové množiny ukladaním a načítavaním zo súborov. Užívateľ si potom jednoducho cez hlavné menu bude môcť zvoliť a načítať trénovaciu množinu, pri čom bude môcť nastaviť dva hlavné parametre. Jedným z nich je proporcia medzi trénovacou a testovacou časťou t.j. koľko percent z načítanej množiny sa má použiť pri procese trénovania a koľko percent pri procese testovania siete. Testovacie dáta sieť pred tým nikdy nevidela a slúžia ako dobrý ukazovateľ, či sa sieť naučila danú úlohu.

V prípade, že užívateľ zvolí rozdelenie 0% pre trénovaciu množinu, prípadne 100% pre trénovaciu množinu, dátová množina sa nerozdelí, ale trénovacia a testovacia časť budú identické t.j. trénovacia množina bude zároveň slúžiť ako testovacia.

Druhým hlavným parametrom, ktorý môže užívateľ nastaviť bude prípad, ak zvolí načítanie dátovej množiny, ktorá vystupuje ako jedno dlhé slovo napr. časový rad funkcie sínus. V takomto prípade bude mať užívateľ možnosť zvoliť maximálnu dĺžku sekvencií, na ktoré bude dlhé slovo rozdelené a bude môcť vynútiť, aby tieto sekvencie postupne narastali na dĺžke. Tým bude implementovaná jedna z heuristík, ktoré uľahčia naučenie siete.

Po úspešnom načítaní a zareprezentovaní dátovej množiny vo vnútorných štruktúrach aplikácie sa odomknú možnosti na trénovanie a testovanie siete.

3.2.4 Nastavenie trénovacích parametrov a trénovanie

Ak už bude mať používateľ vytvorenú sieť aj načítanú dátovú množinu, ktorú si vhodne rozdelil, dostane sa do fázy trénovania. Tu si bude môcť vybrať zo štyroch ponúkaných algoritmov:

- Backpropagation
- Backpropagation Through Time
- Realtime Recurrent Learning
- ESN

Ako sme už zdôraznili v teoretickej časti, nedá sa každým algoritmom dosiahnuť vždy naučenie siete. Preto bude na používateľovi, aby vhodne zvolil algoritmus (alebo si nechal poradiť od cvičiaceho).

Po voľbe algoritmu bude treba nastaviť rýchlosť učenia a počet epôch, počas ktorého má učenie bežať. Zároveň si bude môcť používateľ zvoliť zapnutie „zarážky“, ktorá bude kontrolovať chybu siete

po každej epoche a ak chyba klesne pod nastavenú hodnotu, trénovanie bude prerušené, hoci ešte neprešiel nastavený počet epôch.
Ako posledné bude možné zapnúť a vypnúť heuristiky zlepšujúce učenie ak sú:

- Dávkové učenie
- Rozpad váh
- Momentum
- Klesanie rýchlosti učenia

Pri každej bude treba nastaviť aj jej hodnotu (okrem dávkového učenia), ktorá bude v rozsahu od 0 po 1, hoci sa môžu niekedy tieto veličiny udávať aj v percentách.

Počas trénovania bude môcť používateľ sledovať chybu každej epochy, ktorá sa bude vypisovať do konzoly. Po dokončení učenia sa v grafickom rozhraní zobrazí graf, ktoré lepšie ukáže vývoj chyby počas učenia. Ak nebude používateľ spokojní s konečnou chybou, bude môcť opäť spustiť trénovanie a sieť bude pokračovať v učení od poslednej epochy, čiže nedôjde k žiadnemu resetovaniu siete. To sa bude dať zabezpečiť iba vytvorením novej siete.

V prípade zadania nulového počtu epôch či rýchlosti učenia sa nestane nič, resp. sa sieť nebude nič učiť iba bude bežať nastavený počet epôch.

3.2.5 Testovanie siete

Používateľ bude môcť zvoliť testovanie siete ihneď po vybratí a načítaní dátovej množiny. Je to pre prípad, že si pred tým načíta nejakú natrénovanú sieť uloženú v súbore a bude chcieť na nej overiť schopnosť generalizácie, či predikcie na úplne nových dátach, ktoré získal, alebo si vytvoril.

Bežnejším prípadom bude testovanie siete až po jej natrénovaní. Vtedy používateľ jednoducho zvolí akciu testovania a v konzole sa zobrazí podrobný výpis testovania a v grafickom rozhraní bude vykreslený graf vývoja testovacej chyby. Ak nastane možnosť vizualizovať výstup siete a trénovací signál, bude môcť používateľ zapnúť dodatočné okno, kde sa práve toto zobrazenie ukáže. Pôjde o prípady kedy je výstup jednorozmerný a číselný napr. pri predikcii funkcií ako sínus či zložitejších časových radov.

3.2.6 Uloženie a načítanie siete

Trénovanie siete môže byť zdĺhavá činnosť zaberajúca aj niekoľko desiatok minút, či dokonca hodín. Preto okrem možnosti vytvorenia vždy novej nenatrénovanej siete dostane používateľ aj

možnosť ukladania a načítavania natrénovaných sietí, čo sa určite uplatní pri trénovaní ťažkých úloh na väčších sieťach, kedy sa vynára potreba natrénovanú sieť zachovať a prípadne ďalej použiť, či trénovať. Túto možnosť bude môcť používateľ nájsť v hlavnom menu.

3.3 Vstupy a výstupy

3.3.1 Vstupy zo súborov

Vstupmi zo súborov budeme riešiť načítavanie dátových množín a načítavanie uložených sietí. Pri oboch formátoch sa budeme snažiť, aby boli čitateľné aj pri bežnom otvorení napr. v notepade a ľahko editovateľné. Zároveň bude aplikácia ošetrená na nekorektné názvy súborov, či chýbajúce prípony.

V prípade načítania rovnakého formátu, ktorý však neobsahuje želané dáta (obsahuje niečo úplne iné), dokáže aplikácia používateľa na to upozorniť a korektne sa vrátiť do pôvodného stavu.

3.3.2 Vstupy z grafického rozhrania

Vstupmi grafického rozhrania budeme rozumieť všetky nastaviteľné hodnoty, ktoré aplikácia ponúka. Najčastejšie pôjde o nastavenie čísla, zapnutie alebo vypnutie nejakej dodatočnej funkcie, výber z množiny ponúk.
Väčšina takýchto vstupov bude mať svoje prednastavené hodnoty a ak sa stane, že ich používateľ nenastaví, či už omylom alebo úmyselne, aplikácia sa vždy bude chovať bezchybovo a korektne.

Budeme sa snažiť eliminovať všetky možnosti porušenia integrity vstupov a ak už dôjde k zadaniu nekorektného vstupu, aplikácia to oznámi v konzolovom rozhraní. Jediným kritickým miestom, kde by mohlo prísť k zadaniu zlého vstupu je vytváranie siete. Konkrétne zadávanie veľkostí vrstiev, kde je neakceptovateľná 0, či záporné čísla. Preto budú tieto hodnoty ostro strážené a nepostúpia ďalej ak budú nekorektné.

3.3.3 Výstup do súborov

Naša aplikácia bude podporovať kompletné ukladanie naučenej siete, ktorú uloží pod menom, ktoré zadá používateľ. Ukladanie bude možné ihneď po vytvorení, resp. načítaní siete a to v dvoch

formách.

- Ak máme novovytvorenú sieť, ktorá ešte nebola uložená bude nutné ju „uložiť ako“ a vytvoriť nový súbor.
- Sieť, ktorá už uložená bola, prípadne bola načítaná sa bude dať „uložiť“ a to bez opýtania prepíše jej súbor.

Aplikácia nebude implicitne ponúkať možnosť editovať, či vytvárať dátové množiny, avšak v balíku s ňou bude distribuovaná aj sada utilít, ktoré dokážu niektoré typy úloh ľahko vytvoriť. Zdrojové kódy utilít bude môcť hocikto editovať a nebude na to potrebovať výrazné programátorské schopnosti. Podľa časových možností vytvoríme snáď aj lepšie a inteligentnejšie nástroje na tvorbu dátových množín, ale nebude to náš primárny cieľ.

3.3.4 Výstup do grafického rozhrania a konzoly

Jedným z hlavných cieľov, ktoré sme si stanovili bola dobrá vizualizácia výsledkov, o čo sa bude starať grafické rozhranie a zároveň konzola. Kvôli lepšiemu prehľadu budú vždy výstupy v textovej forme pridávané do konzoly a kvôli predstavivosti a vizuálnemu vnemu zobrazované v grafickom rozhraní.

Pôjde o výpis vývoja chyby siete počas učenia a zároveň jeho zobrazenie v grafe. V textovej podobe bude mať používateľ lepšiu predstavu a veľkosti chyby a v grafe zas uvidí, v ktorých časoch a ako veľmi chyba klesala, prípadne sa takmer držala na jednej úrovni.

Obdobnú stratégiu budeme voliť aj pri testovaní siete. Taktiež bude simultánne výstup zobrazovaný do konzoly aj do grafického rozhrania. V konzole si bude môcť používateľ overiť, aký bol výstup siete a k nemu zodpovedajúci signál, kde ihneď uvidí, či sa sieť naučila danú úlohu alebo nie. Graf bude zobrazovať krivku testovacej chyby na každej vzorke dát. Naviac bude umožnená aj vizualizácia výstupu siete, avšak tu bude treba ošetriť, aby sa vizualizácia vytvárala iba pri povolených prípadoch s jedným numerickým výstupom. Ináč by mohlo prísť ku kritickému zlyhaniu celej aplikácie.

3.4 Technologické riešenie

Celá aplikácia bude bežať na platforme Linux. Jej vývoj bude prebiehať pod distribúciou Ubuntu. O konverzii pre operačný systém Windows budeme možno uvažovať v budúcnosti. Jazykom, v ktorom bude aplikácia naprogramovaná bude Python verzia 3.1. Vývoj kódu bude prebiehať v

prostredí NetBeans.

Na dizajnovanie a implementáciu GUI bude použitá knižnica GTK. Túto knižnicu sme zvolili vzhľadom na jej jednoduché použitie v Pythone, keďže existuje jej kompilácia v tomto prostredí pod názov PyGTK. Taktiež existuje paleta nástrojov na jednoduché vyskladanie GUI a jeho modifikovanie. Konkrétne bude použitý RAD tool (rapid application development tool) Glade Interface Designer.

Knižnica PyBrain [9] bude zabezpečovať implementáciu štruktúr neurónových sietí, algoritmu BP a štruktúr zabezpečujúcich reprezentáciu dátových množín. Táto knižnica je voľne šíriteľná a dostupná z internetu. Bočným produktom našej práce by malo byť rozšírenie tejto knižnice o algoritmy na učenie rekurentných sietí.

Knižnica NumPy obsahuje metódy na manipuláciu s použitými matematickými štruktúrami ako sú polia, matice, permutácie množín, náhodne generované čísla. Dnes už zväčša býva súčasťou Pythonu. Taktiež je voľne stiahnuteľná z internetu.

3.5 Časti aplikácie

Logicky je možné návrh aplikácie rozčleniť na 4 skupiny, pri čom každá bude obsahovať triedy, ktoré budú mať spoločné rysy alebo budú vykonávať funkcie so spoločným cieľom.

3.5.1 Jadro aplikácie a grafické rozhranie

Táto skupina bude obsahovať tri triedy, ktoré budú obsluhovať interakciu užívateľa s aplikáciou, riadiť jej hlavný beh a reagovať na udalosti, ktoré budú aplikácii zasielané.

Inicializačná trieda bude slúžiť ako rozhranie medzi triedou grafického rozhrania (GUI) a základnou triedou neurónových sietí. Primárne pôjde o predávanie akcií prichádzajúcich z jedného aj druhého smeru. Zároveň inicializačná trieda obsahuje aj hlavú slučku.

Úlohou GUI bude pri inicializácii rozparsovať a vytvoriť grafické okno a všetky jeho ovládacie prvky načítaním súboru GUI.glade vytvoreného za pomoci nástroja Glade Interface Designer. Ďalej sa bude starať o odosielanie akcií prevedených používateľom a na základe ich kontextu modifikovať okno, otvárať a zatvárať dialógy, renderovať grafické výstupy. GUI trieda by mala predstavovať zároveň prvú hrádzu, ktorá zachytáva všetky nekorektné vstupy. Používateľ by mal zvonka prichádzať do kontaktu iba s touto triedou.

Trieda neurónovej siete bude skrytá za inicializačnou triedou a bude komunikovať iba s ňou.

Táto trieda na základe interakcie užívateľa s GUI bude budovať a modifikovať dátový model aktuálnej neurónovej siete. Bude zapuzdrovať triedu štruktúry neurónovej siete, trénovaciu triedu a triedu dátovej množiny. Zároveň bude zhromažďovať vizualizačné dáta (napr. priebeh chyby z trénovania) a posielať ich cez inicializačnú triedu späť ku GUI a tá ich bude zobrazovať.

3.5.2 Triedy štruktúr

Skupina bude obsahovať tri triedy, ktoré vytvárajú spomínané modely sietí. S týmito troma triedami bude komunikovať trieda neurónovej siete z jadra aplikácie a podľa potreby bude volať ich konštruktor, či ostatné metódy.

Trieda `ElmanNetwork` bude obsahovať konštruktor, v ktorom sa pomocou metód knižnice PyBrain vytvorí Elmanova sieť. Vstupom konštruktora budú dimenzie jednotlivých vrstiev, typy aktivačných funkcií skrytej a výstupnej vrstvy a voľba pripojenia prahovej jednotky k týmto vrstvám. Bude ešte obsahovať metódu na rozvinutie siete pre časový rad dĺžky T, ktorá bude vracať doprednú sieť.

Trieda `JordanNetwork` sa bude podobať triede implementujúcej Elmanovu sieť, akurát ináč inicializuje rekurentné prepojenie. Taktiež bude obsahovať metódu na rozvinutie siete, ktorá z nej vytvorí doprednú sieť a priradí ju k jej návratovej hodnote.

Trieda `EchostateNetwork` bude obsahovať konštruktor podobný predchádzajúcim dvom triedam a vytvorí sieť s echo stavmi. V argumentoch konštruktoru pribudne hodnôt spektrálneho rádiusu, ktorá sa posunie do inicializácie rezervoárového spojenia.
Trieda `ReservoirConnection` bude implementovať rezervoárové spojenie, pri čom bude mať implicitne danú riedkosť matice váh a v argumente dostane spektrálny rádius.

3.5.3 Trénovacie triedy

Trénovacie triedy budú obsahovať implementácie trénovacích algoritmov. S jednotlivými triedami bude komunikovať trieda neurónovej siete, ktorá bude volať ich metódy pri procese trénovania. Ich inicializácia bude nanovo prebiehať pri každom spustení trénovania.

Trieda `BPTT_Trainer` bude implementovať BPTT algoritmus, pri čom bude obsahovať konštruktor a verejnú metódu trénovania počas jednej epochy. V argumente konštruktora dostane

všetky potrebné parametre učenia. Tie ktoré nedostane budú automaticky nastavené na predvolené hodnoty, ktoré sa väčšinou rovnajú vypnutiu. Vnútorne bude obsluhovať vytváranie rozvinutých sietí a následné aplikácie zmien váh na materskú sieť.

Trieda `RTRL_Trainer` bude implementovať algoritmus RTRL. Taktiež bude obsahovať konštruktor a metódu spúšťajúcu trénovanie na jednu epochu. Konštruktor dostáva zoznam parametrov trénovania, ktoré budú obsluhované podobne ako pri triede `BPTT_Trainer`. Ostatné metódy budú privátne a budú implementovať jednotlivé kroky algoritmu.

Trieda `ESN_Trainer` bude implementovať jednokrokový trénovací algoritmus pre echo state siete spomínaný v teoretickej analýze. Bude obsahovať konštruktor dostávajúci zoznam parametrov učenia a metódu učenia. Implicitne bude rozdeľovať trénovaciu množinu na dve podmnožiny, ktoré sa použijú na nabudenie a natrénovanie siete.

3.5.4 Rozhranie dátových množín

Bude obsahovať iba jednu triedu, ktorá bude mať za úlohu načítať a rozparsovať dátovú množinu zo súboru. Na základe hlavičky, ktorú načíta, zareprezentuje všetky dáta vhodným spôsobom, aby mohli prichádzať do trénovacích algoritmov a siete. Bude obsahovať vnútorné pravidlá, ktoré musia byť striktne dodržané pri tvorbe dátových množín a tak sa zabráni načítaniu nezmyselného obsahu. Taktiež bude implementovať metódu, ktorá bude rozdeľovať dátovú množinu na trénovaciu a testovaciu, pri čom bude zachovávať poradie jednotlivých sekvencií.

3.6 Dokumentácia

Okrem zdrojových kódov hlavnej aplikácie a utilít bude k balíku pridružená aj krátka dokumentácia v podobe manuálu v anglickom jazyku. Manuál bude obsahovať opisy krok za krokom ako pracovať s aplikáciou. Ako v nej vytvoriť požadovanú sieť, načítať dátovú množinu, trénovať a testovať sieť. Bude taktiež obsahovať návod ako vytvoriť vlastnú trénovaciu množinu s podrobným popisom pravidiel, ktorými sa náš formát riadi. Na konci budú obsiahnuté riešenia nami predpripravených úloh v podobe krátkych tutoriálov. V nich budú približné parametre siete a učiacich algoritmov, aby sa daná sieť dokázala naučiť konkrétny problém.

3.7 Predpripravené úlohy

Rozhodli sme sa vytvoriť niekoľko úloh ilustrujúcich možnosti rekurentných sietí. Pôjde o úlohy asociačné a predikčné.

Jedna skupina úloh bude vygenerovaná Mealyho automatmi a pôjde o presne také isté automaty, ako sa spomínajú na prednášku z neurónových sietí, alebo v odporúčanej literatúre k tomuto predmetu [5].

Druhá skupina úloh bude pozostávať z periodických a chaotických časových radov. Taktiež sme sa snažili pokryť prípady, ktoré sa spomínajú na prednáške.

Treťou skupinou sú úlohy, ktoré budú vytvorené pre naše potreby, hlavne testovanie a debugovanie algoritmov. Budú to jednoduché úlohy, pozostávajúce z naučenia logických funkcií AND, XOR v rôznych variantoch.

4 Implementácia aplikácie

4.1 Jadro a GUI

`__init__.py`

Súbor obsahuje zavádzaciu triedu, ktorá sa vykoná po spustení v konzole príkazom:

```
python __init__.py
```

Zavádzacia trieda obsahuje inštanciu grafického rozhrania, ktorý sa ihneď načíta a vytvorí, a inštanciu triedy spravujúce aktuálnu neurónovú sieť. Táto trieda sa inicializuje na `None`.

`GUI.py`

Pomocou nástroja Glade Interface Designer bolo vytvorené grafické prostredie aplikácie. Výstupom Glade je XML súbor, ktorý sa pri inicializácii načíta a builder knižnice pyGTK ho rozparsuje a vytvorí tak ovládacie prvky a okno.

Následne je zavedená tabuľka prepájajúca signály GUI s odpovedajúcimi funkciami, ktoré prevažne komunikujú s jadrom aplikácie.

`neuralNet.py`

Obsahuje triedu, ktorá je dátovou štruktúrou reprezentujúcou neurónovú sieť, s ktorou sa aktuálne pracuje. Pri inicializácii (t.j. po zadaní povelu na vytvorenie siete) zavolá konštruktor jedného z troch typov ponúkaných sietí, prípadne ak načíta zadaný súbor a vytvorí sieť, ktorá je v ňom uložená. Trénovací algoritmus aj dátovú množinu inicializuje na `None`.

Obsahuje tri verejné metódy. Metóda `onLoadDataset` po zavolaní hlavnou triedou vytvorí inštanciu triedy, ktorá načíta zo súboru a zareprezentuje dátovú množinu. Následne ju rozdelí na trénovaciu a testovaciu časť. Metóda `onTrain` vytvorí inštanciu trénovacieho algoritmu s parametrami poslanými z GUI cez hlavnú triedu. Následne *n*-krát volá trénovaciu metódu algoritmu, kde *n* je počet epoch. Po natrénovaní vyšle požiadavku GUI o vykreslenie grafu chyby z pozbieraných údajov za jednotlivé epochy. Metóda `onTest` vykonáva testovanie siete. Meria strednú kvadratickú chybu (mean squared error, MSE) a vyšle požiadavku GUI na vykreslenie grafu chyby pri testovaní.

4.2 Štrukturálne triedy

`ElmanModel.py, JordanModel.py`

Obsahuje triedu `ElmanNetwork` (`JordanNetwork`), ktorá sa inicializuje pri zavolaní z neuralNet. Táto trieda je podtriedou `RecurrentNetwork`, implementovanej v knižnici PyBrain. Sieť sa vyskladá z blokov. Každý blok predstavuje jednu vrstvu alebo jedno spojenie medzi vrstvami. Tieto stavebné kamene sú implementované v knižnici PyBrain. Vyskladanie siete funguje podľa pevného modelu, kde je jedna vstupná, jedna skrytá, jedna kontextová a jedna výstupná vrstva. Každá vrstva sa vytvára s nastavenou aktivačnou funkciou a počtom neurónov. Oba parametre vstupujú do konštruktora ako jeho argumenty.

Metóda `unfoldNetwork` vykonáva rozvinutie siete s parametrom T, ktorý hovorí o dĺžke rozvinutia. Je vytvorená inštancia triedy `FeedForwardNetwork`, ktorá reprezentuje sieť s dopredným šírením. Buduje sa obdobným spôsobom ako sa buduje sieť pri inicializácii. Navyše sa volá privátna metóda `_copyWeights`, ktorá skopíruje všetky váhové spojenia z vlastnej (rekurentnej) siete do rozvinutej doprednej siete, podľa špecifikácie rozvíjaného modelu.

Metóda `foldNetwork` robí opak ako robila predošlá metóda, hoci to na prvý pohľad nemusí byť jasné. Táto metóda zráta derivácie všetkých spojení rozvinutej doprednej siete a posiela ich v návratovej hodnote. Následne sa tieto derivácie aplikujú na zmenu váh rekurentnej siete, avšak to už v algoritme BPTT.

`EchoStateNetwork.py`

Obsahuje rovnomennú triedu, ktorá pri svojej inicializácii buduje obdobným procesom sieť s echo stavmi. Taktiež je pevne určený model s jednou vstupnou, skrytou a výstupnou vrstvou a s jednou kontextovou vrstvou.

`ReservoirConnection.py`

Obsahuje triedu `ReservoirConnection`, ktorá je podtriedou triedy `FullConnection` implementovanej v knižnici PyBrain, a reprezentuje riedke spojenie medzi kontextovou vrstvou a rezervoárom v triede `EchoStateNetwork`. Po inicializácii je matica prepojení zriedená na 20% a jej spektrálny rádius je nastavený na hodnotu, aká je zaslaná z GUI.

4.3 Trénovacie algoritmy

Algoritmus BP je priamo implementovaný v knižnici PyBrain.

`BPTT_Trainer.py`

Implementuje algoritmus BPTT, pri čom je podtriedou triedy `BackpropTrainer` implementovanej v PyBrain. Jeho činnosť približuje nasledujúci pseudokód:

```
for s in dataset.sequences:
      #získanie vstupu a signálu učiteľa dátovej sekvencie s
      i,t = s
      #rozvinutie siete na dĺžku vstupu
      unfoldedNetwork = network.unfoldNetwork(len(i))
      #výpočet derivácii a zároveň váženej chyby na počet
      #výstupných neurónov rozvinutej siete pre sekvenciu s
      e,p = self._calcDerivs(unfoldedNetwork, s)
      errors += e
      ponderation += p
      # získanie derivácií pre úpravu rekurentnej siete
      derivs = network.foldNetwork(len(i))
      # výpočet aktuálneho gradientu siete
      gradient = derivs - self.weightdecay * network.params
      # výpočet a aplikácia nových váh na základe gradientu
      new = self.descent(gradient)
      network.params[:] = new
```

`RTRL_Trainer.py`

Obsahuje implementáciu algoritmu RTRL. Triedu `RTRL_Trainer` je podtriedou triedy Trainer, ktorá reprezentuje základnú triedu pre všetky trénovacie algoritmy implementovanú v PyBrain. Trénovací proces RTRL by sme z algoritmického hľadiska mohli zhrnúť na dva hlavné kroky. Algoritmus pracuje s maticou váh a maticou parciálnych derivácii každého stavového neurónu podľa každej váhy. V prvom kroku je treba vypočítať maticu parciálnych derivácií pre nasledujúci čas t+1 a v druhom kroku je treba vykonať update váhovej matice na základe získanej chyby a matice parciálnych derivácií.

Opäť pre lepšiu predstavu uvádzame nasledujúce zjednodušenie našej implementácie v podobe pseudokódu:

```
for s in dataset.sequences:
      for i, t in s:
            # vytvorenie váhovej matice W z váh siete
            self._initWeightMatrix()
            # aktivácia siete na vstupe i
            self.module.activate(i)
            # skopírovanie výstupov všetkých neurónov do množiny z
            self._copyOutputs()
            # výpočet váženej chyby na počet výstupných neurónov
            e, p = self._calcError(t)
            errors += e
            ponderation += p
            #  výpočet  matice  parciálnych  derivácií  podľa  rovnice
            # (2.24)
            self.P = self._calcPartialDerivs()
            # update matice váh podľa matice P a chyby E
            self._updateWeightMatrix()
            # prepis matice váh do váh siete
            self._applyWeightMatrix()
            self.t += 1
```

ESN_Trainer.py

V triede ESN_Trainer implementuje trénovací algoritmus pre ESN siete. Dá sa rozdeliť na tri kroky. V prvom kroku sa aktivuje sieť na 30% trénovacej množiny. V druhom kroku dochádza k vzorkovaniu (ukladaniu aktivácií neurónov rezervoára a signálu učiteľa) a v treťom kroku k výpočtu inverznej matice a následne k výpočtu výstupných váh. Opäť pripájame aj pseudokód našej implementácie:

```
# rozdelenie trénovacej množiny na časť, ktorá iba aktivuje sieť a na časť,
ktorá je použitá pri trénovaní
sam, trn = self._splitDataset(0.3)
# aktivovanie siete na množine sam
for seq in sam.sequences:
      for input, _ in seq:
            self.module.activate(input)
# vytvorenie matice aktivácií rezervoára a stĺpcového vektora so signálom
```

```
učiteľa
self.M=matrix(len(trn),self.module['hidden0'].dim)
self.T=arange(0.,len(trn))
i = 0
for seq in trn.sequences:
      for input, target in seq:
            # aktivácia siete a ukladanie aktivácií rezervoára
            self.module.activate(input)
            self.M[i,:] = self.module['hidden0']
            # ukladanie signálu učiteľa do vektora T
            self.T[i] = target
            i += 1
# výpočet matice výstupných váh pomocou súčinu pseudoinverznej matice k
matici M a vektora T
Wout = pinv(self.M) * self.T
# prepis vypočítanej výstupnej matice Wout do váh siete
self._applyWeights(Wout)
```

4.4 Rozhranie dátových množín

`DataSet.py`

Obsahuje triedu DataSet, ktorá spracúva vstupy v podobe súborov vo formáte XML a následne ich vhodne zareprezentuje do interných dátových štruktúr. Inicializuje sa pri načítaní dátovej množiny a kedy vytvorí prázdne premenné slúžiace na uchovávanie parametrov (`options`), samotných dát (`dataset`), pomocnej pamäte pre úschovu dát (`_tempset`) a konverznej tabuľky (`conversion`). Metóda load načítava súbor s dátami, ktorého meno dostane ako argument. Používa XMLhandler z knižnice PyBrain. Pomocou neho rozparsuje načítaný súbor a vyhľadá v ňom koreňový tag `<PyBrain>`, ktorý musí byť obsiahnutý v každom súbore. Následne vytvorí konverznú tabuľku ak nájde tag `<Conversion>`. Z neho načíta znak a jeho one-hot kódovanie. Z tagu `<DataSet>` načíta všetky hodnoty do options a ďalej sa riadi podľa nich. Ak je dátová množina zadefinovaná ako sekvenčná, premennej dataset priradí inštanciu triedy `SequentialDataSet` implementovanej v PyBrain, v opačnom prípade vytvorí `SupervisedDataSet`. Podľa toho vyhľadá tag `<SequentialDataSet>` prípadne `<SupervisedDataSet>`, ktorý hovorí o type vstupov a výstupov.

Rozoznávame tri typy vstupov/výstupov:

- `Numeric` – jeden číselný vstup
- `MultiNumeric` – číselný vektor
- `NonNumeric` – znakový vstup, ktorý sa zakóduje konverznou tabuľkou

Pre sekvenčné dáta sa ešte načítava údaj (`SequentialDistribution`) určujúci, či sa má sekvencia v súbore rozdeliť podľa toho ako je v ňom uložená, alebo sa má načítať ako jedna dlhá sekvencia. Vtedy sa uloží do dočasnej pamäte _tempset a na konci načítania sa náhodne rozdelí (podľa voľby aj v poradí s narastajúcou dĺžkou).
Posledným krokom je načítanie dát, ktoré sú predchádzajúcimi parametrami jasne špecifikované.

Metóda `splitDataset` rozdeľuje dátovú množinu na dve časti podľa zadaného koeficientu z intervalu <0,1>. Zachováva poradie sekvencií, ktoré je často kľúčové pre naučenie úlohy. Z tohto dôvodu bola táto metóda vytvorená, keďže PyBrain rovnakú metódu obsahuje, lenže tá poradie sekvencií nezachováva.

4.5 Utility pre tvorbu úloh

Spolu s aplikáciou sme vyvinuli aj dve utility na jednoduchšiu tvorbu vybraných úloh.

`Automaton.py`

Je utilita určená na simuláciu automatov s konečným stavom. Obsahuje abecedu `E`, tabuľku prechodov medzi stavmi `T`, ktorej modifikáciou dosiahne užívateľ simulovanie nového automatu. Popri tom treba nastaviť premennú options, z ktorej sa bude vytvárať hlavička súboru popísaná v predošlej podkapitole. Procedúra `generate_words` vytvára náhodne slová z abecedy `E`. Dá sa nastaviť počet slov, ich maximálna dĺžka a premenlivosť dĺžky. V tele utility si jednoduchými úpravami môže užívateľ vytvoriť hocijaký typ úlohy. Tieto úpravy ponechávame na šikovnosti užívateľa, pretože prečítaním zdrojového kódu na ne príde za pár minút.

`Timeseries.py`

Druhá utilita je určená na načítavanie textového súboru, ktorý na každom riadku obsahuje jeden číselný údaj. Primárne táto utilita vytvára predikčné úlohy z časových radov. Obsahuje procedúru load, ktorá načíta daný textový súbor a uloží ho do premennej dataset. Po zavolaní procedúry `save` sa

premenná dataset uloží vo formáte XML, pričom opäť treba nastaviť premennú options, rovnako ako v predchádzajúcej utilite. Dáta v takomto textovom formáte nie je na internete problém zohnať a takto ich možno ľahko skonvertovať a používať v našej aplikácii.

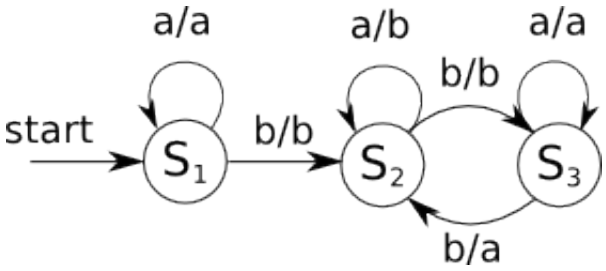
4.6 Vytvorené úlohy

Asociačná úloha s trojstavovým Mealyho automatom:

Úloha demonštruje schopnosť siete naučiť sa asociovať vstupné slová automatu s výstupnými, pri čom si sieť vytvorí vlastnú reprezentáciu stavov. Sieť dostáva na vstup slová po jednotlivých symboloch a výstupom je symbol prislúchajúci výstupu automatu v danom stave.

Predikčná úloha s trojstavovým Mealyho automatom:

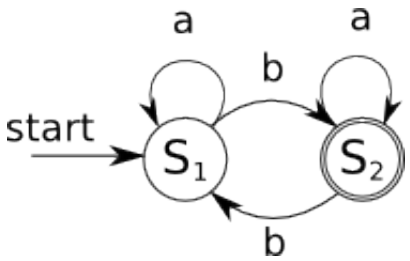
Sieť na vstupe dostáva jednotlivé symboly slov a má predikovať nasledujúci symbol. Očakáva sa podobné správanie siete ako v predošlých úlohách.



Obr. 7: Schéma použitého automatu s troma stavmi a vstupnými / výstupnými symbolomi

Klasifikačná úloha s dvojstavovým Mealyho automatom:

V tejto úlohe dostáva sieť na vstupe slová a na konci každého slova má rozhodnúť, či slovo patrí alebo nepatrí do gramatiky automatu. Opäť si sieť musí vytvoriť vnútornú reprezentáciu stavov automatu. Ide o úlohu, kedy sieť nemá k dispozícii signál učiteľa vždy, ale iba v niektorých časových okamihoch.



Obr. 8: Schéma použitého automatu s dvoma stavmi a vstupnými symbolmi a akceptačným stavom S_2

Predikčná úloha s časovým radom:

Úloha je zameraná na predikciu nasledujúcej hodnoty časového radu. Sieť dostáva číselnú hodnotu radu v čase *t* a má predikovať hodnotu v čase *t*+1, pričom by mala vypozorovať periódu, amplitúdu a iné charakteristiky. Prvá úloha je naučenie sa funkcie sínus, ktorá má pravidelnú periódu a jednoduchý priebeh. Na demonštrovanie výpočtovej sily sú potom zamerané úlohy s chaotickými časovými radmi Mackey-Glass, slnečné škvrny a dáta laseru.

5 Záver

Touto prácou sme sa snažili prispieť ku skvalitneniu výučby umelých neurónových sietí. Predovšetkým chceme ponúknuť študentom jednoduchý nástroj, s ktorým by mohli v praxi prechádzať preberané úlohy a hlavne nadobudnúť skúsenosti v práci s rekurentnými sieťami.

Snažili sme sa nájsť optimum medzi dostatočnou všeobecnosťou didaktického produktu a časovou zvládnuteľnosťou jeho naprogramovania. Nevyhli sme sa určitým hraniciam, v rámci ktorých sa študent môže pohybovať. No príliš veľká všeobecnosť aplikácie by sa takmer rovnala samotnému programovaniu, keďže každý nový model si vyžaduje svoje špecifiká. Dúfame, že v tomto smere sme uspeli.

Modulárnosť zdrojového kódu a jednoduchá možnosť aktualizácie ponúka hneď niekoľko nápadov na vylepšenia, či rozšírenia aplikácie. Preto by sme nechceli vývoj ukončiť v tomto časovom okamihu, ale určite pokračovať ďalej. Hlavne prispôsobiť aplikáciu čo najviac potrebám študentov. Akékoľvek postrehy majú možnosť zasielať na emailovú adresu uvedenú v dokumentácii. Do budúcnosti sa črtá niekoľko smerov, ktorými by sme chceli program rozšíriť a podľa úspešnosti snáď aj distribuovať na iné univerzity.

6 Použitá literatúra

[1] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Internal Representations by Error Propagation. In: J.L. McClelland, D.E. Rumelhart, and PDP Research Group, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations, chapter 8, MIT Press, Cambridge, 1986.

[2] P.J. Werbos. Backpropagation Through Time: What It Is and How to Do It. Proceedings of the IEEE, 10: 1550-1560, 1990.

[3] R.J. Williams and D. Zipser. Experimental Analysis of the Real-Time Recurrent Learning Algorithm. Connection Science, 1: 87-111, 1989.

[4] S. Haykin. Neural Neworks: a Comprehensive Foundation, Prentice Hall. , 1999.

[5] V. Kvasnička, L. Beňušková, J. Pospíchal, I. Farkaš, P. Tiňo, and A. Kráľ. Úvod do teórie neurónových sietí, IRIS, Bratislava, 1997, ISBN 80-88778-30-1

[6] H. Jaeger. Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach. GMD Report 159, Fraunhofer Institute AIS, 2002.

[7] M. Lukoševičius and H. Jaeger. Overview of reservoir recipes. Technical Report No. 11, Jacobs University Bremen, 2007.

[8] A. Waibel, T. Hanazawa, G. Hinton and K. Shikano, In "Phoneme recognition using time-delay neural networks", IEEE trans. ASSP 37: 328-339, 1989.

[9] T. Schaul et al.: PyBrain. Journal of Machine Learning Research 1 (2010) 999-1000