Istraživanje kvarkovsko-gluonske strukture protona pomoću strojnog učenja

Ćorić, Ivan

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

Ivan Ćorić

Istraživanje kvarkovsko-gluonske strukture protona pomoću strojnog učenja

Diplomski rad

Zagreb, 2019.

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

INTEGRIRANI PREDDIPLOMSKI I DIPLOMSKI SVEUČILIŠNI STUDIJ
FIZIKA; SMJER ISTRAŽIVAČKI

Ivan Ćorić

Diplomski rad

Istraživanje kvarkovsko-gluonske strukture protona pomoću strojnog učenja

Voditelj diplomskog rada: prof. dr. sc. Krešimir Kumerički

Ocjena diplomskog rada: _____

Povjerenstvo: 1. _____

2. _____

3. _____

Datum polaganja: _____
Zagreb, 2019.

Zahvaljujem se mentoru na pruženoj pomoći pri izradi rada te svojoj obitelji na potpori tijekom studija.

Sažetak

U ovom radu promatrat će se mogućnost ekstrakcije komptonskih form faktora (CFF funkcija) iz tipičnih opservabla mjerenih u procesima leptoprodukcije fotona, gdje su CFF funkcije modelirane neuronskim mrežama. Kako bi se provjerilo hoće li postupak učenja CFF funkcija neuronskim mrežama biti uspješan i o čemu ovisi kvalitetna ekstrakcija, generirani su umjetni eksperimentalni podaci (opservable na koje je dodan šum) iz poznatog Goloskokov-Kroll modela za CFF funkcije. Pokazano je da se uz dovoljan broj podataka, CFF funkcije mogu naučiti, da je kvaliteta ekstrakcije to bolja što ima više podataka te su uočene opservable čije uključivanje u skup podataka za učenje poboljšava kvalitetu ekstrakcije u većoj mjeri nego uključivanje drugih opservabli.

Ključne riječi: Generalizirane partonske distribucije; Duboko virtualno komptonsko raspršenje; Komptonski form faktori; Neuronske mreže.

Research of quark-gluon structure of proton using machine learning

Abstract

In this thesis we will study a possibility of extraction of Compton form factors (CFFs) from typical observables measured in the process of leptoproduction of photon, where the CFF functions will be modeled with neural networks. In order to verify the validity of learning CFF functions using neural networks and to determine what makes a quality extraction, artificial (*mock*) experimental data (observables to which noise was applied) was generated from popular Goloskokov-Kroll model for CFF functions. It is shown that with the enough data, CFF functions can be learned, that the quality of extraction is getting better when increasing the number of data and some observables were noticed which had more effect on the quality of extraction than the others.

Keywords: Generalized parton distributions; Deeply Virtual Compton Scattering; Compton form factors; Neural networks.

Sadržaj

1 Uvod

Kvantna kromodinamika (engl. *Quantum Chromodynamics*, QCD) područje je fizike koje proučava jake interakcije između kvarkovskih sustava koje skupno nazivamo hadronima. Zasnovana je na ne-Abelovoj baždarnoj teoriji polja te je u režimu niskih energija neperturbativna, zbog čega još uvijek nemamo dovoljno kvalitetno poznavanje hadrona.

Hadrone dijelimo na sustave kvarka i antikvarka ($q\bar{q}$) koje nazivamo mezonima te sustave od tri kvarka (qqq) ili tri antikvarka ($\bar{q}\bar{q}\bar{q}$) koje nazivamo barionima. Kod opisa hadrona kvarkovima uvodimo novi kvantni broj, *boju*, koji izlazi iz $SU(3)$ simetrije teorije. Interakcija između kvarkova ide preko bezmasenih baždarnih vektorskih bozona spina 1, koje nazivamo gluonima.

Prije pojave QCD teorije, hadroni su opisivani modelom u kojem oni imaju podstrukturu te su građeni od drugih fermiona i bozona koje skupno nazivamo partonima. Kao jedan od uspjeha QCD teorije smatramo partonske distribucijske funkcije (engl. *Parton Distribution Functions*, PDFs). Proučavanjem PDF-ova dobivamo saznanja o stupnjevima slobode partona. Glavni proces u kojem proučavamo PDF-ove naziva se duboko neelastično raspršenje (engl. Deep Inelastic Scattering, DIS).

Nakon PDF-ova, kao alat za proučavanje partona uvedene su generalizirane partonske distribucije (engl. *Generalized Parton Distributions*, GPDs), kojima možemo proučavati trodimenzionalne stupnjeve slobode kvarkova i gluona. U određenom limesu one postaju stare partonske distribucijske funkcije. GPD-ovi se mogu proučavati u eksperimentima poput leptoprodukcije ili fotoprodukcije fotona, leptonskih parova i mezona. U eksperimentima ne mjerimo direktno GPD-ove već tzv. komptonske form faktore (engl. *Compton Form Factors*, CFFs) koji su zapravo integrali GPD funkcija.

U ovom radu promatrat će se mogućnost ekstrakcije komptonskih form faktora (CFF-ova) iz tipičnih opservabla mjerenih u procesima leptoprodukcije fotona, gdje su CFF-ovi modelirani neuronskim mrežama. Kao model za CFF funkcije su izabrane neuronske mreže iz razloga što ne unose nikakvu pristranost pri izboru modela.

U drugom poglavlju dan je teorijski uvod u kojem su definirane GPD funkcije te CFF funkcije koje želimo modelirati, također opisan je proces leptoprodukcije fotona te su opisane tipične opservable mjerene u takvim procesima. U trećem poglavlju dana je inspiracija za nastanak prvog umjetnog neurona, opisan je umjetni neuron te na koji način se umjetni neuroni kombiniraju u potpuno povezane neuronske mreže. Objašnjen je algoritam propagacije unazad te je iskazan teorem univerzalne aproksimacije za neuronske mreže. U četvrtom poglavlju dani su rezultati ekstrakcija CFF funkcija iz različitih opservabli.

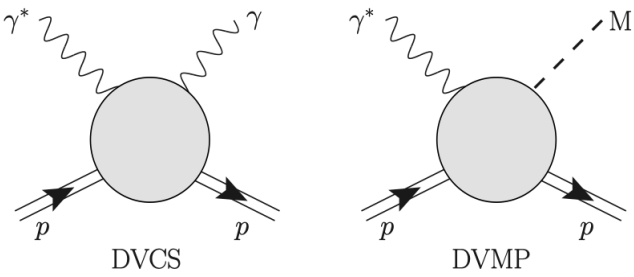
2 Teorijski uvod

2.1 Generalizirane partonske distribucije

Generalizirane partonske distribucije (GPD-ovi) uvedene su 1994. [1] te ponovno, nezavisno otkrivene 1997. [2], [3]. Ubrzo im se, u okviru kvantne kromodinamike (QCD), počinje davati velik značaj te se počinju intenzivno proučavati zbog njihovih jedinstvenih svojstva poput: direktne veze sa spinom hadrona, povezanosti s drugim neperturbativnim objektima koji su bili već dobro proučeni (partonske distribucijske funkcije (PDF-ovi) i form faktori (FF-ovi)) te činjenice da su GPD-ovi direktno povezani s matričnim elementom tenzora energije-impulsa QCD-a ugniježđenog između hadronskih stanja.

U sustavu beskonačnog impulsa, partonske distribucijske funkcije predstavljaju distribucije longitudinalnog impulsa partona unutar hadrona, dok su form faktori Fourierove transformacije distribucije naboja hadrona u transverzalnoj ravnini. GPD-ovi, dakle, nose informacije o trodimenzionalnim distribucijama partona u hadronu. Uz to što GPD-ovi obuhvaćaju oba ta koncepta, obuhvaćaju i koncept distributcijske amplitude (DA) za slučaj piona.

Tenzor energije-impulsa, koji se uobičajeno proučavao putem gravitacije, sada se može proučavati kroz elektromagnetsko raspršenje. Naime, zbog mogućnosti faktorizacije perturbativnih i neperturbativnih (QCD) doprinosa, ekskluzivna elektroprodukcija realnog fotona ili mezona je najčišći pristup GPD-ovima pa onda i samom tenzoru energije-impulsa.



Slika 2.1: Prikaz DVCS (lijevo) i DVMP (desno) procesa. Posuđeno iz [5].

GPD-ovima se pristupa putem procesa Dubokog virtualnog komptonskog raspršenja (DVCS) te procesa Duboke virtualne produkcije mezona (DVMP) koji su prikazani na slici 2.1. Kroz te eksperimente nemamo direktan pristup GPD-ovima, već pristupamo integralima GPD-ova i određenih jezgri (tzv. komptonskim form faktorima, CFF-ovima). S eksperimentalne strane, mjerenje se pokazalo zahtjevnim jer zahtijeva veliki luminozitet i rezoluciju. Unatoč tome, istraživanja su pokazala mogućnost

izvođenja DVCS mjerenja nakon čega su provedeni brojni eksperimenti, kao što je opisano u [5].

Uvedimo notaciju kao u [5]. Neka je metrički tenzor dan kao:

$$g_{\mu\nu} = g^{\mu\nu} = \mathrm{diag}(1, -1, -1, -1). \tag{2.1}$$

Za normalizaciju Levi-Civita tenzora $\varepsilon^{\mu\nu\rho\sigma}$ uzima se $\varepsilon^{0123} = 1$. Definirajmo transverzalnu projekciju Levi-Civita tenzora kao $\varepsilon_\perp^{\mu\nu} \equiv \varepsilon^{\mu\nu-+}$, čije jedine neiščezavajuće komponente su:

$$\varepsilon_\perp^{12} = -\varepsilon_\perp^{21} = \varepsilon_{12}^\perp = -\varepsilon_{21}^\perp = 1. \tag{2.2}$$

Za svaki četvero-vektor a definirajmo koordinate svjetlosnog stošca:

$$a^\pm = \frac{1}{\sqrt{2}}\left(a^0 \pm a^3\right) \quad \text{i} \quad a = \left(a^+, \mathbf{a}, a^-\right), \tag{2.3}$$

gdje je skalarni produkt u tom sustavu dan kao:

$$(ab) = a^+b^- + a^-b^+ - \mathbf{a}\cdot\mathbf{b}. \tag{2.4}$$

Kod elemenata hadronske matrice $\langle P_2|O|P_1\rangle$, 1 predstavlja ulazno, a 2 izlazno stanje. Za ukupni impuls P i ukupni prijenos impulsa Δ vrijedi:

$$\begin{aligned} P &= P_1 + P_2, \\ \Delta &= P_2 - P_1. \end{aligned} \tag{2.5}$$

Također za Mandelstamovu t varijablu vrijedi $t = \Delta^2$. Uvedimo varijablu asimetrije (engl. *skewness*) η kao:

$$\eta = -\frac{\Delta^+}{P^+}, \tag{2.6}$$

dok će sa simbolom ξ biti označena varijabla približno jednaku $x_B/\left(2 - x_B\right)$, gdje je x_B uobičajena Bjorkenova varijabla skaliranja. S M će biti označena masa protona, a s e_q naboj čestice q u jedinicama naboja pozitrona $|e|$. θ je Heavisideova step funkcija, γ_μ Diracova matrica, $\sigma_{\mu\nu} = i[\gamma_\mu, \gamma_\nu]/2$, a s Q označavamo virtualnost fotona. Za velik Q^2 dominantni doprinos dolazi od operatora s najmanjim tzv. twistom. Twist τ predstavlja razliku između dimenzije operatora te njegovog spina $\tau = d - j$.

Sada možemo definirati generalizirane partonske distribucije. U nepolariziranom (vektorskom) sektoru GPD-ovi H i E su definirani kao:

$$\frac{h^+}{P^+}H^q + \frac{e^+}{P^+}E^q = \int \frac{dz^-}{2\pi} e^{ixP^+z^-} \left\langle P_2 \left| \overline{q}(-z)\gamma^+ q(z) \right| P_1 \right\rangle\Big|_{z^+=0,\mathbf{z}=\mathbf{0}}, \tag{2.7}$$

$$\frac{h^+}{P^+}H^g + \frac{e^+}{P^+}E^g = \frac{4}{P^+}\int \frac{dz^-}{2\pi} e^{ixP^+z^-} \left\langle P_2 \left| G_a^{+\mu}(-z) G_{a\mu}^+(z) \right| P_1 \right\rangle \Big|_{z^+=0,\mathbf{z}=\mathbf{0}}, \tag{2.8}$$

a u polariziranom (aksijalno-vektorskom) sektroru GPD-ovi $\tilde{H}$ i $\tilde{E}$ definirani su kao:

$$\frac{\tilde{h}^+}{P^+}\tilde{H}^q + \frac{\tilde{e}^+}{P^+}\tilde{E}^q = \int \frac{dz^-}{2\pi} e^{ixP^+z^-} \left\langle P_2 \left| \overline{q}(-z)\gamma^+\gamma_5 q(z) \right| P_1 \right\rangle \Big|_{z^+=0,\mathbf{z}=\mathbf{0}}, \tag{2.9}$$

$$\frac{\tilde{h}^+}{P^+}\tilde{H}^g + \frac{\tilde{e}^+}{P^+}\tilde{E}^g = \frac{4}{P^+}\int \frac{dz^-}{2\pi} e^{ixP^+z^-} \left\langle P_2 \left| G_a^{+\mu}(-z) i\epsilon_{\mu\nu}^{\perp} G_a^{\nu+}(z) \right| P_1 \right\rangle \Big|_{z^+=0,\mathbf{z}=\mathbf{0}}, \tag{2.10}$$

gdje su:

$$\begin{aligned} h^\mu &= \overline{u}(P_2)\gamma^\mu u(P_1); \quad e^\mu = \frac{i\Delta_\nu}{2M}\overline{u}(P_2)\sigma^{\mu\nu}u(P_1), \\ \tilde{h}^\mu &= \overline{u}(P_2)\gamma^\mu\gamma_5 u(P_1); \quad \tilde{e}^\mu = \frac{\Delta^\mu}{2M}\overline{u}(P_2)\gamma_5 u(P_1), \end{aligned} \tag{2.11}$$

a spinori su normalizirani na $\overline{u}(p)\gamma^\mu u(p) = 2p^\mu$. Za najniži, twist dva, možemo definirati još dodatna četiri GPD-a H_T, E_T, $\tilde{H}_T$ i $\tilde{E}_T$ u tenzorskom sektoru:

$$\begin{aligned} &\int \frac{dz^-}{2\pi} e^{ixP^+z^-} \left\langle P_2 \left| \overline{q}(-z) i\sigma^{+i} q(z) \right| P_1 \right\rangle \Big|_{z^+=0,z=0} = \frac{1}{P^+} \\ &\times \overline{u}(P_2)\left[H_T^q i\sigma^{+i} + E_T^q \frac{\gamma^+\Delta^i - \gamma^i\Delta^+}{2M} + \tilde{H}_T^q \frac{P^+\Delta^i - P^i\Delta^+}{2M^2} + \tilde{E}_T^q \frac{\gamma^+P^i - \gamma^iP^+}{2M} \right] u(P_1) \end{aligned}, \tag{2.12}$$

gdje je $i = 1,\ 2$. U gluonskom sektoru imamo analogno:

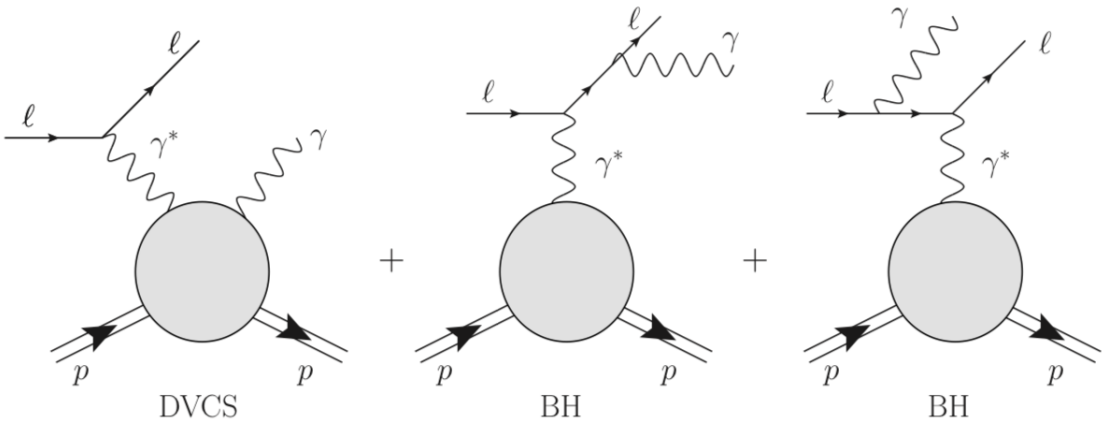
$$\begin{aligned} &\frac{4}{P^+}\int \frac{dz^-}{2\pi} e^{ixP^+z^-} \left\langle P_2 \left| \mathcal{S} G_a^{+i}(-z) G_a^{j+}(z) \right| P_1 \right\rangle \Big|_{z^+=0,\mathbf{z}=\mathbf{0}} = \mathcal{S}\frac{1}{P^+}\frac{P^+\Delta^j - \Delta^+P^j}{2MP^+} \\ &\times \overline{u}(P_2)\left[H_T^g i\sigma^{+i} + E_T^g \frac{\gamma^+\Delta^i - \gamma^i\Delta^+}{2M} + \tilde{H}_T^g \frac{P^+\Delta^i - P^i\Delta^+}{2M^2} + \tilde{E}_T^g \frac{\gamma^+P^i - \gamma^iP^+}{2M} \right] u(P_1), \end{aligned} \tag{2.13}$$

gdje $\mathcal{S}$ označava simetrizaciju i oduzimanje traga za nekontrahirane indekse. Kao što je opisano u [4], iz Lorentz invarijantnosti slijedi da GPD-ovi ovise samo o tri varijable: x, η i t. Na primjer imamo $H^q = H^q(x,\eta,t)$ te analogno za ostale GPD-ove.

Kao primjer graničnog slučaja GPD-ova možemo pogledati situaciju kada $P_1 = P_2$, kada se GPD-ovi svode na obične PDF-ove:

$$\begin{aligned} H^q(x,0,0) &= \theta(x)q(x) - \theta(-x)\overline{q}(-x), \\ H^g(x,0,0) &= \theta(x)xg(x) - \theta(-x)xg(-x), \\ \tilde{H}^q(x,0,0) &= \theta(x)\Delta q(x) + \theta(-x)\Delta\overline{q}(-x) \text{ i} \\ \tilde{H}_T^q(x,0,0) &= \theta(x)\Delta_T q(x) - \theta(-x)\Delta_T\overline{q}(-x), \end{aligned} \tag{2.14}$$

gdje su $q(x)$ i $g(x)$ obični, $\Delta q(x)$ polarizirani, a $\Delta_T q(x)$ transverzalni (engl. *transversity*) PDF-ovi. Osim ovih svojstava, GPD-ovi zadovoljavaju druga različita korisna svojstva (pravila sume, polinomijalnost, pozitivnost) kao što je opisano u [5], no u njih nećemo ulaziti.



Slika 2.2: Prikaz DVCS i Bethe-Heitler procesa. Posuđeno iz [5].

2.2 Leptoprodukcija realnog fotona

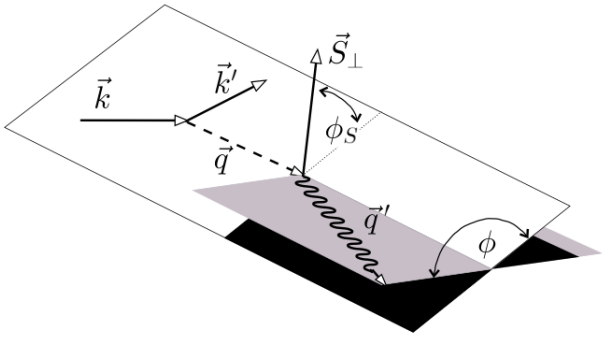
Leptoprodukcija realnog fotona je proces u kojem imamo pristup gore spomenutim komptonskim form faktorima (CFF-ovima). To je proces u kojem imamo interferenciju amplitude dubokog virtualnog komptonskog raspršenja (DVCS-a) te tzv. Bethe-Heitler amplitude kao što je prikazano na slici 2.2. Kao što je opisano u [5], opći udarni presjek je dan kao:

$$\frac{d^5\sigma}{dx_B dQ^2 d|t| d\phi d\phi_S} = \frac{\alpha^3 x_B}{16\pi^2 Q^4 \sqrt{1+\epsilon^2}} |\mathcal{T}|^2, \tag{2.15}$$

gdje je α konstanta fine strukture, $\epsilon = 2x_B M/Q$, kutevi ϕ i ϕ_S su prikazani na slici 2.3, a $\mathcal{T}$ predstavlja koherentnu superpoziciju DVCS i Bethe-Heitler amplituda:

$$|\mathcal{T}|^2 = |\mathcal{T}_{\mathrm{BH}} + \mathcal{T}_{\mathrm{DVCS}}|^2 = |\mathcal{T}_{\mathrm{BH}}|^2 + |\mathcal{T}_{\mathrm{DVCS}}|^2 + \mathcal{I}, \tag{2.16}$$

gdje je $\mathcal{I} = \mathcal{T}_{\mathrm{DVCS}}\mathcal{T}_{\mathrm{BH}} + \mathcal{T}_{\mathrm{BH}}\mathcal{T}_{\mathrm{DVCS}}$ interferencijski član.



Slika 2.3: Definicije momenata i kuteva važnih za leptoprodukciju realnog fotona u Trento konvenciji. Posuđeno iz [5].

U ovom radu baziramo se na ekstrakciji CFF funkcija. Naime, $\mathcal{T}_{\mathrm{DVCS}}$ se može rastaviti u kompleksne komptonske form faktore (CFF-ove):

$$\mathcal{H}, \mathcal{E}, \tilde{\mathcal{H}}, \tilde{\mathcal{E}}, \mathcal{H}_{\mathrm{T}}, \mathcal{E}_{\mathrm{T}}, \tilde{\mathcal{H}}_{\mathrm{T}}, \tilde{\mathcal{E}}_{\mathrm{T}},$$

koji se mjere u eksperimentima. Kroz član $|\mathcal{T}_{\mathrm{DVCS}}|^2$ pristupa se umnoščima CFF funkcija, dok kroz $\mathcal{I}$ čistom realnom i imaginarnom dijelu CFF funkcija do na Bethe-Heitler amplitudu, ali neperturbativni dio Bethe-Heitler amplitude $\mathcal{T}_{\mathrm{BH}}$ dobro je poznat za promatrano kinematičko područje. Komptonski form faktori definirani su kao u [5]:

$$\mathcal{F}(\eta, t) = \sum_q e_q^2 \int_{-1}^{1} dx \left[\frac{1}{\eta - x - i\epsilon} - \frac{1}{\eta + x - i\epsilon} \right] F^q(x, \eta, t) \tag{2.17}$$

i

$$\tilde{\mathcal{F}}(\eta, t) = \sum_q e_q^2 \int_{-1}^{1} dx \left[\frac{1}{\eta - x - i\epsilon} + \frac{1}{\eta + x - i\epsilon} \right] \tilde{F}^q(x, \eta, t), \tag{2.18}$$

gdje je $\mathcal{F} = \mathcal{H},\ \mathcal{E},\ ...$, i respektivno $F^q = H^q,\ E^q, ...$ predstavlja odgovarajuću generaliziranu partonsku distribuciju. Možemo primjetiti da su komptonski form faktori relativno jednostavniji od GPD-ova, ovise samo o dvije varijable (jedna se izgubi u integraciji), ali opet su kompleksne funkcije pa se sastoje od realnog i imaginarnog dijela.

Opservable koje mjerimo u ovom procesu dobivaju se različitim izborima polarizacije i naboja snopa i mete te određenom vrstom harmonijske analize. Prateći [5], navedene su različite opservable koje se mogu mjeriti u DVCS procesu te iz kojih se dobivaju podaci o CFF funkcijama. Udarni presjek za leptoprodukciju realnog fotona leptonom l, koji ima naboj e_l (u jedinicama naboja pozitrona) i helicitet $h_l/2$, na nepolariziranoj meti dan je kao:

$$d\sigma^{h_l, e_l}(\phi) = d\sigma_{\mathrm{UU}}(\phi) \left[1 + h_l A_{\mathrm{LU,DVCS}}(\phi) + e_l h_l A_{\mathrm{LU,I}}(\phi) + e_l A_{\mathrm{C}}(\phi)\right], \tag{2.19}$$

gdje konvencionalno prvi indeks označava polarizaciju snopa (U kao nepolarizirano (engl. *unpolarized*), L kao longitudinalna polarizacija, T kao transverzalna polarizacija), a drugi polarizaciju mete. Ako postoji mogućnost longitudinalne polarizacije snopa i pozitivno i negativno nabijenih snopova, navedene asimetrije naboja snopa (A_{C}) i asimetrije spina snopa ($A_{\mathrm{LU,I}}$ i $A_{\mathrm{LU,DVCS}}$) se mogu izolirati:

$$A_{\mathrm{C}}(\phi) = \frac{(d\sigma^{\overset{+}{\rightarrow}} + d\sigma^{\overset{+}{\leftarrow}}) - (d\sigma^{\overset{-}{\rightarrow}} + d\sigma^{\overset{-}{\leftarrow}})}{4d\sigma_{\mathrm{UU}}(\phi)}, \tag{2.20}$$

$$A_{\mathrm{LU,I}}(\phi) = \frac{(d\sigma^{\overset{+}{\rightarrow}} - d\sigma^{\overset{+}{\leftarrow}}) - (d\sigma^{\overset{-}{\rightarrow}} - d\sigma^{\overset{-}{\leftarrow}})}{4d\sigma_{\mathrm{UU}}(\phi)} \quad \text{i} \tag{2.21}$$

$$A_{\mathrm{LU,DVCS}}(\phi) = \frac{(d\sigma^{\overset{+}{\rightarrow}} - d\sigma^{\overset{+}{\leftarrow}}) + (d\sigma^{\overset{-}{\rightarrow}} - d\sigma^{\overset{-}{\leftarrow}})}{4d\sigma_{\mathrm{UU}}(\phi)}. \tag{2.22}$$

Ovdje $+$ ($-$) označavaju pozitivan (negativan) naboj snopa, a $\rightarrow$ ($\leftarrow$) označava desni (lijevi) helicitet.

Ako ne postoji mogućnost mijenjanja naboja snopa (kao u npr. *Jefferson Lab*), tada, uz naboj snopa e_l, možemo izolirati sljedeću asimetriju spina snopa:

$$A_{\mathrm{LU}}^{e_l}(\phi) = \frac{d\sigma^{\overset{e_l}{\rightarrow}} - d\sigma^{\overset{e_l}{\leftarrow}}}{d\sigma^{\overset{e_l}{\rightarrow}} + d\sigma^{\overset{e_l}{\leftarrow}}}. \tag{2.23}$$

Nadalje, ako imamo nepolarizirani snop te longitudinalno polariziranu metu,

$$A_{\mathrm{UL}}^{e_l}(\phi) = \frac{[d\sigma^{\leftarrow^{\overset{e_l}{\Rightarrow}}} + d\sigma^{\rightarrow^{\overset{e_l}{\Rightarrow}}}] - [d\sigma^{\leftarrow^{\overset{e_l}{\Leftarrow}}} + d\sigma^{\rightarrow^{\overset{e_l}{\Leftarrow}}}]}{[d\sigma^{\leftarrow^{\overset{e_l}{\Rightarrow}}} + d\sigma^{\rightarrow^{\overset{e_l}{\Rightarrow}}}] + [d\sigma^{\leftarrow^{\overset{e_l}{\Leftarrow}}} + d\sigma^{\rightarrow^{\overset{e_l}{\Leftarrow}}}]}, \tag{2.24}$$

gdje s $\Rightarrow$ ($\Leftarrow$) označavamo paralelnu (anti-paralelnu) polarizaciju mete s obzirom na impuls snopa. Za longitudinalno polarizirani snop te longitudinalno polariziranu metu imamo sljedeću opservablu:

$$A_{\mathrm{LL}}^{e_l}(\phi) = \frac{[d\sigma^{\rightarrow^{\overset{e_l}{\Rightarrow}}} + d\sigma^{\leftarrow^{\overset{e_l}{\Leftarrow}}}] - [d\sigma^{\leftarrow^{\overset{e_l}{\Rightarrow}}} + d\sigma^{\rightarrow^{\overset{e_l}{\Leftarrow}}}]}{[d\sigma^{\rightarrow^{\overset{e_l}{\Rightarrow}}} + d\sigma^{\leftarrow^{\overset{e_l}{\Leftarrow}}}] + [d\sigma^{\leftarrow^{\overset{e_l}{\Rightarrow}}} + d\sigma^{\rightarrow^{\overset{e_l}{\Leftarrow}}}]}. \tag{2.25}$$

Za transverzalno polariziranu metu te pozitivno i negativno nabijene snopove imamo:

$$A_{\mathrm{UT,I}}(\phi, \phi_S) = \frac{d\sigma^+(\phi_S) - d\sigma^+(\phi_S + \pi) + d\sigma^-(\phi_S) - d\sigma^-(\phi_S + \pi)}{d\sigma^+(\phi_S) + d\sigma^+(\phi_S + \pi) + d\sigma^-(\phi_S) + d\sigma^-(\phi_S + \pi)} \quad \text{te} \tag{2.26}$$

$$A_{\mathrm{UT,DVCS}}\left(\phi,\phi_S\right)=\frac{d\sigma^{+}\left(\phi_S\right)-d\sigma^{+}\left(\phi_S+\pi\right)-d\sigma^{-}\left(\phi_S\right)+d\sigma^{-}\left(\phi_S+\pi\right)}{d\sigma^{+}\left(\phi_S\right)+d\sigma^{+}\left(\phi_S+\pi\right)+d\sigma^{-}\left(\phi_S\right)+d\sigma^{-}\left(\phi_S+\pi\right)}. \tag{2.27}$$

Kod slučaja eksperimenata u kojima se ne mogu mjeriti direktno udarni presjeci već samo asimetrije, kako bi se dobila približno linearna ovisnost o komptonskim form faktorima, može se koristiti dominacija Bethe-Heitler člana u nazivniku. Na primjer za prvi sinusni harmonik asimetrije spina snopa:

$$A_{LU}^{-,\sin\phi}\equiv\frac{1}{\pi}\int_{-\pi}^{\pi}d\phi\sin\phi A_{LU}^{-}(\phi), \tag{2.28}$$

za koji vrijedi da je približno proporcionalan kombinaciji CFF-ova kao:

$$A_{LU}^{-,\sin\phi}\propto\mathfrak{Im}\left(F_1\mathcal{H}-\frac{t}{4M^2}F_2\mathcal{E}+\frac{x_B}{2}\left(F_1+F_2\right)\tilde{\mathcal{H}}\right). \tag{2.29}$$

U ovom radu korišteni su još i nepolarizirani te polarizirani udarni presjek. Nepolarizirani udarni presjek (engl. *beam spin sum*) je definiran kao

$$d\sigma=d\sigma^{\rightarrow}+d\sigma^{\leftarrow}, \tag{2.30}$$

tj. kao zbroj udarnih presjeka s desnim i lijevim helicitetom. Polarizirani udarni presjek (engl. *beam spin difference*) definiran je kao:

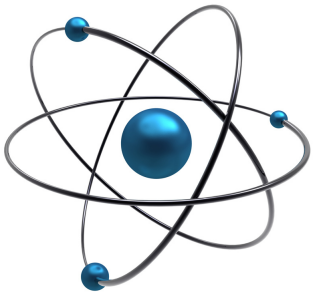
$$\Delta\sigma=\frac{1}{2}[d\sigma^{\rightarrow}-d\sigma^{\leftarrow}], \tag{2.31}$$

tj. kao razlika udarnih presjeka s desnim odnosno lijevim helicitetom.

3 Neuronske mreže

3.1 Motivacijski uvod

Umjetne neuronske mreže (nadalje samo neuronske mreže) su računalni sustavi koji su nastali pokušajem imitiranja bioloških neuronskih mreža kakve nalazimo u čovjeku odnosno životinjama. Biološke neuronske mreže su sposobne naučiti kako kvalitetno obaviti određen zadatak bez da su prije programirane za to. Na primjer, na slici 3.1 se nalazi pojednostavljeni model atoma te ga čovjek (biološka neuronska mreža) prepozna relativno brzo. Biološka neuronska mreža taj posao prvotno nije znala obaviti, nego se kroz različite susrete s tim modelom atoma izmijenila na način da uspješno prepozna atom na slici. To svojstvo učenja obavljanja kompleksnih zadataka je vrlo poželjno u različitim granama znanosti i ljudi kroz povijest su ga pokušali imitirati.



Slika 3.1: Prikaz pojednostavljenog modela atoma. Posuđeno iz [6].

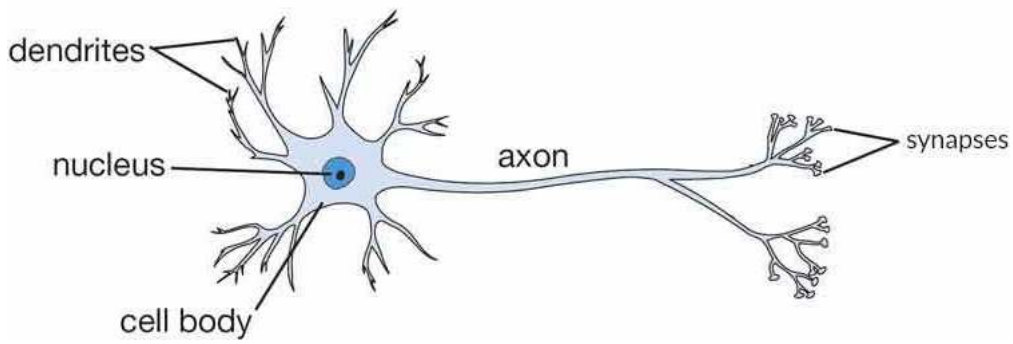
3.2 Povijesni razvoj neuro-računarstva

1943. godine W. McCulloch i W. Pitts napravili su matematički model neurona. Kako je računalna moć tada bila slaba prva praktična ostvarenja tog modela dogodila su se tek u kasnim 70-im godinama. D. O. Hebb je 1949. godine postavio hipotezu učenja kao metaboličku promjenu u neuronima, koja je temelj razvoja mehanizma učenja neuronskih mreža. 1951. M. Minsky i D. Edmonds su stvorili prvu neuronsku mrežu (engl. *Stohastic Neural Analog Reinforcment Computer - SNARC*) koja simulira traženje izlaza iz labirinta.

F. Rosenblatt je 1958. godine je stvorio perceptron, algoritam za raspoznavanje uzoraka koji je pokazivao sposobnost učenja. 1959. godine M. Minsky i J. McCarthy osnivaju Laboratorij za umjetnu inteligenciju (engl. *AILab*). 1969. godine entuzijazam nestaje zbog tzv. problema pridruživanja odgovornosti (engl. *Credit assignment*

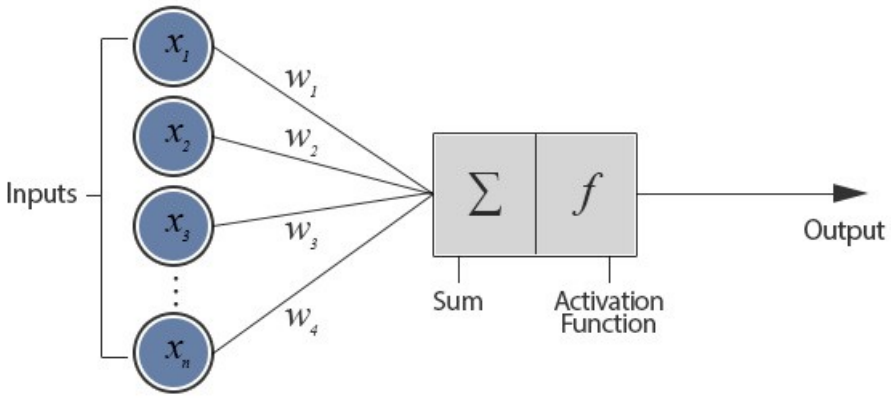
problem): "Kako odrediti koliko je svaki procesni element odgovoran za pogrešku na izlazu?" Također, problemi su i s nedostatkom procesne moći za veće neuronske mreže te činjenicom da se linearno neodvojivi problemi ne mogu riješiti tadašnjim neuronskim mrežama. 1982. J. Hopfield popularizira (otkrivena ranije 1974. godine) tzv. Hopfieldovu mrežu. Konačno 1986. Rumelhart, Hinton i Williams ponovno otkrivaju i populariziraju rješenje problema pridruživanja odgovornosti (engl. *Credit assignment problem*) tj. algoritam propagacije unazad (engl. *Back error propagation - Backpropagation*). Nakon devedesetih godina do danas događa se značajan porast istraživanja u tom području.

3.3 Usporedba biološkog i umjetnog neurona



Slika 3.2: Prikaz biološkog neurona. Posuđeno iz [7].

Kao što je rečeno prije, umjetni neuroni su inspirirani biološkim neuronima. Na slici 3.2 se nalazi shema biološkog neurona. Biološki neuron prima ulazne signale preko dendrita. Signali prolazeći kroz tijelo neurona bivaju obrađeni te preko aksona izlaze iz neurona.



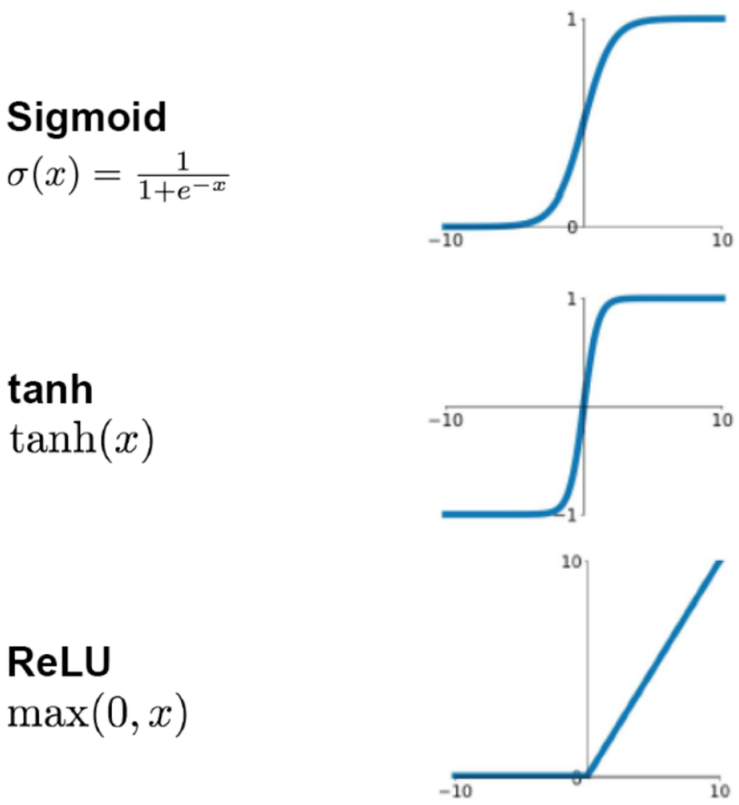
Slika 3.3: Prikaz umjetnog neurona. Posuđeno iz [8].

Umjetni neuron, prikazan na slici 3.3, će biti imitacija biološkog neurona. Ulaze

umjetnog neurona označimo s $x_1,\ x_2,\ ...,\ x_n$, gdje je n broj ulaza u neuron. Podrazumijevamo da su ulazi u neuron realni brojevi. Označimo s $w_1,\ w_2,\ ...,\ w_n$ tzv. težine neurona. Težine neurona predstavljaju "težinu" koju pripisujemo odgovarajućem ulazu (što je težina i veća to je važniji ulaz i). Ulazi umjetnog neurona pomnoženi s odgovarajućom težinom, zbrajaju se te im se dodaje b tzv. prag (engl. *bias*). Da bi neuronske mreže mogle opisivati i nelinearne funkcije, na tu linearnu kombinaciju djelujemo nekom nelinearnom funkcijom f. Ako izlaz neurona označimo s y, formula koja opisuje umjetni neuron glasi:

$$y = f\Big(\sum_{i=1}^{n} w_i x_i + b\Big). \tag{3.1}$$

Nelinearnu funkciju f nazivamo aktivacijskom funkcijom. Ulogu aktivacijske funkcije mogu igrati različite nelinearne funkcije poput: sigmoide, tangensa hiperbolnog, zglobnice (engl. *Rectified Linear Unit - ReLU*) i druge. Vizualizacije različitih aktivacijskih funkcija dane su na slici 3.4.



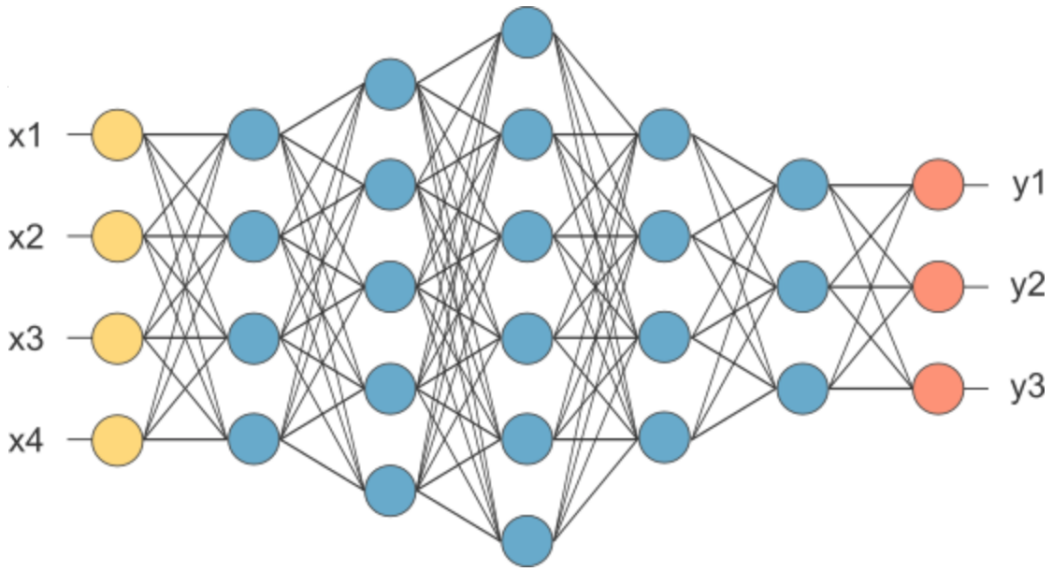
Slika 3.4: Prikaz različitih aktivacijskih funkcija. Posuđeno iz [9].

Važno je napomenuti da je biološki neuron bio samo inspiracija za model prvog neurona. Danas se zajednica odmaknula od uspoređivanja umjetnih neuronskih mreža s biološkim neuronskim mrežama. Prepoznato je da umjetne neuronske mreže

dobro obavljaju posao za koji su namijenjene (obrada slike, obrada prirodnog teksta, obrada zvuka, regresija...) i za to se koriste, dok je veza s biološkim neuronskim mrežama većinom povijesna.

3.4 Umjetne neuronske mreže

Umjetni neuroni se nadalje povezuju u (umjetne) neuronske mreže. Postoje različiti načini povezivanja (različite unaprijedne (engl. *feedforward*) mreže poput potpuno povezanih neuronskih mreža, konvolucijskih neuronskih mreža, autoenkodera; neuronske mreže s povratnom vezom (engl. *Recurrent Neural Networks - RNN*) poput engl. *Long - Short Term Memory* (LSTM) mreže, bidirekcionalne mreže s povratnom vezom (BRNN) i mnogih drugih), no u ovom radu je opisana i upotrijebljena samo potpuno povezana neuronska mreža.



Slika 3.5: Primjer umjetne neuronske mreže. Posuđeno iz [10].

Potpuno povezana neuronska mreža se sastoji od neurona poredanih u slojeve kao na slici 3.5. Označimo s N_i broj neurona u i-tom sloju te neka neuronska mreža ima m slojeva. Neuroni u prvom sloju kao ulaze primaju same ulaze u mrežu (x_1, x_2, ..., x_n). Svaki neuron prvog sloja djeluje svojim težinama $w_1^{(k)}$, $w_2^{(k)}$, ..., $w_n^{(k)}$ na sve ulaze, gdje k predstavlja redni broj neurona unutar sloja. Nakon djelovanja aktivacijske funkcije dobivamo izlaze neurona koje ćemo označiti s $h_1^{(k)}$, gdje je k opet redni broj neurona unutar sloja. Dakle imamo N_1 jednadžbi oblika:

$$h_1^{(k)} = f\Big(\sum_{i=1}^{n} w_i^{(k)} x_i + b_1^{(k)}\Big). \tag{3.2}$$

Ako uvedemo konvenciju da kada aktivacijska funkcija f djeluje na vektor, ona djeluje zasebno na svaku komponentu vektora (engl. *elementwise*), možemo cijeli jedan sloj neuronske mreže kompaktnije zapisati kao:

$$\boldsymbol{h_1} = f\Big(W_1\boldsymbol{x} + \boldsymbol{b_1}\Big), \tag{3.3}$$

gdje je $\boldsymbol{h_1} = (h_1^{(1)},\ h_1^{(2)},\ ...,\ h_1^{(N_1)})^T$, $\boldsymbol{x} = (x_1,\ x_2,\ ...,\ x_n)^T$, $\boldsymbol{b_1} = (b_1^{(1)},\ b_1^{(2)},\ ...,\ b_1^{(N_1)})^T$, a W_1 je matrica težina prvog sloja:

$$W_1 = \begin{pmatrix} w_1^{(1)} & w_2^{(1)} & \dots & w_n^{(1)} \\ w_1^{(2)} & w_2^{(2)} & \dots & w_n^{(2)} \\ \vdots & & \ddots & \\ w_1^{(N_1)} & w_2^{(N_1)} & ... & w_n^{(N_1)} \end{pmatrix}. \tag{3.4}$$

Dakle u prvom sloju smo uveli nN_1 težina te N_1 pragova, sveukupno $(n+1)N_1$ slobodnih parametara.

Drugi sloj možemo sada jednostavnije uvesti koristeći matričnu notaciju kao:

$$\boldsymbol{h_2} = f\Big(W_2\boldsymbol{h_1} + \boldsymbol{b_2}\Big), \tag{3.5}$$

gdje su veličine definirane analogno: $\boldsymbol{h_2}$ i $\boldsymbol{b_2}$ su vektori dimenzije N_2, a W_2 matrica dimenzija $N_2 \times N_1$ te aktivacijska funkcija f djeluje na svaku komponentu vektora. Ovaj sloj ima $(N_1+1)N_2$ slobodnih parametara. Analogno uvodimo sve slojeve do zadnjeg m-tog kod kojeg uvodimo oznaku za izlaz iz mreže $\hat{\boldsymbol{y}} = (\hat{y}_1,\ \hat{y}_2,\ ...,\ \hat{y}_{N_m})$, gdje je N_m broj neurona u zadnjem sloju koji je određen dimenzionalnošću funkcije koja se želi aproksimirati. Jednadžba zadnje sloja izgleda kao:

$$\hat{\boldsymbol{y}} = f\Big(W_m\boldsymbol{h_{m-1}} + \boldsymbol{b_m}\Big). \tag{3.6}$$

Cijelu neuronsku mrežu kao funkciju F od ulaza $\boldsymbol{x}$ možemo zapisati kao:

$$\hat{\boldsymbol{y}} = F_{w,b}(\boldsymbol{x}) \tag{3.7}$$

gdje su w i b skupovi svih težina odnosna pragova. Takva neuronska mreža ima ukupno

$$(n+1)N_1 + (N_1+1)N_2 + \ ... + (N_{m-1}+1)N_m \tag{3.8}$$

slobodnih parametara. Na primjer, neuronska mreža sa slike 3.5 ima 136 slobodnih parametara koje predstavljaju težine i pragovi.

Ovakva neuronska mreža može imati više primjena. Može se koristiti za klasifika-

ciju te za regresiju.

Klasifikacija je postupak određivanja kojoj klasi (grupi) ulazni podatak pripada. Ako su ulazni podaci dvodimenzionalna (siva) slika koju razvučemo u jednodimenzionalni vektor možemo pokušati učiti što se nalazi na slici, npr. pas ili mačka. Kod takve (binarne) klasifikacije u zadnjem sloju imali bismo samo jedan neuron, a uz korištenje sigmoide kao aktivacijske funkcije u zadnjem sloju (slika sigmoide je $[0,\ 1]$) izlaz bismo mogli tumačiti kao vjerojatnost pripadanja ulazne slike jednoj od klasa.

Regresija je postupak modeliranja funkcije. Ako želimo modelirati k-dimenzionalne funkcije neuronskim mrežama, trebamo izabrati k-neurona u zadnjem sloju neuronske mreže te pobrinuti se da je aktivacija u zadnjem sloju takva da može pokriti sve vrijednosti funkcije koju modeliramo.

3.5 Algoritam propagacije unatrag

Kako bi neuronska mreža mogla obavljati zadatak za koji je zamišljena mora "naučiti" slobodne parametre (težine i pragove) za svaki pojedini neuron. To učenje se najčešće radi iterativnim, rekurzivnim i efikasnim algoritmom propagacije unatrag. Efikasnost algoritma je posljedica pravila ulančavanja derivacija (engl. *chain rule*). Algoritmom propagacije pogreške unatrag rješava gore spomenuti problem pridruživanja odgovornosti jer se njime može točno odrediti koliko je koji neuron odgovoran za grešku na izlazu.

Bez gubitka općenitosti pretpostavimo da želimo riješiti problem regresije. Neka je dano N ulaznih podataka $D = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}$ gdje i ide od 1 do N, koji predstavljaju uzorak funkcije koju želimo estimirati (naučiti). Definirajmo funkciju gubitka $L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)})$, uz $\hat{\boldsymbol{y}}^{(i)} = F_{w,b}(\boldsymbol{x}^{(i)})$, koja ovisi o predikciji mreže za ulazne podatke $\boldsymbol{x}^{(i)}$ te pravoj vrijednosti funkcije koju želimo naučiti za dani i te neka vrijedi:

$$\begin{aligned} ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}|| > 0 &\rightarrow L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}) > 0, \\ ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}|| = 0 &\rightarrow L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}) = 0, \\ ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}|| > ||\hat{\boldsymbol{y}}^{(j)} - \boldsymbol{y}^{(j)}|| &\rightarrow L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}) > L(\hat{\boldsymbol{y}}^{(j)}, \boldsymbol{y}^{(j)}). \end{aligned} \tag{3.9}$$

Primjer najjednostavnije funkcije gubitka može biti kvadrat odstupanja tj.

$$L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}) = ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||^2. \tag{3.10}$$

Kao ukupnu funkciju gubitka možemo gledati $L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)})$ na samo jednom primjeru i (taj pristup se zove engl. *online* učenje), na pravom podskupu skupa D tj.

$$\tilde{L} = \sum_{i \in S \subset D} L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}), \tag{3.11}$$

(taj pristup se zove engl. *mini-batch* učenje) te na cijelom skupu D:

$$\tilde{L} = \sum_{i \in D} L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}), \tag{3.12}$$

(a ovaj se pristup zove engl. *full-batch* učenje).

Jednom kad odaberemo ukupnu funkciju gubitka i slučajno inicijaliziramo težine i pragove (postoje i pametniji pristupi inicijalizaciji težina i pragove, no ulaženje u to prelazi opseg ovoga rada) možemo započeti učenje. Proces učenja je iterativan proces. U svakom se koraku izračuna funkcija gubitka te je potrebno izračunati kako u svakom koraku ažurirati težine i pragove. Glavna ideja algoritma propagacije pogreške unazad jest u osnovi gradijentni spust (samo napravljen efikasno).

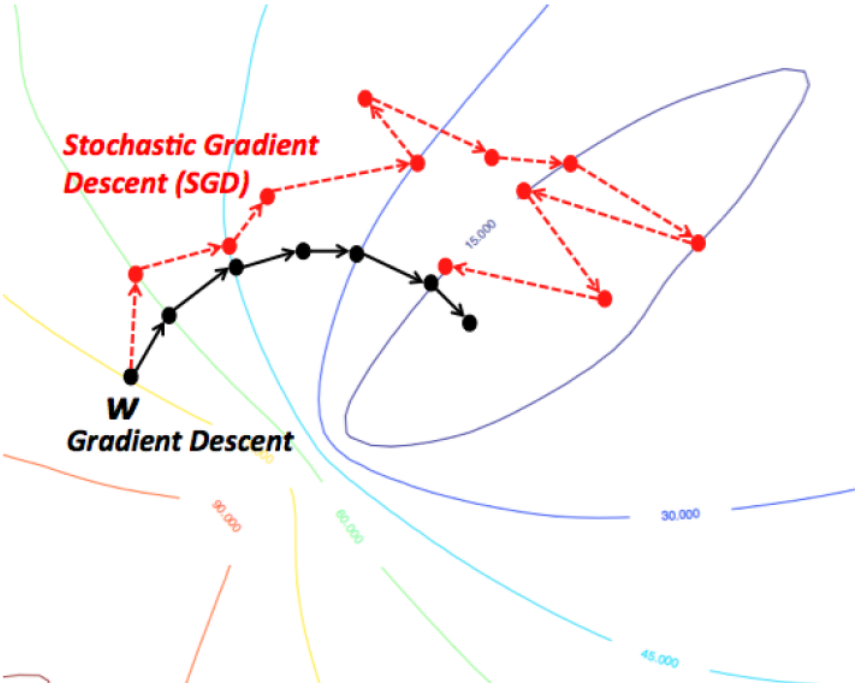
Naime, funkcija gubitka govori koliko je mreža loše odradila posao za koji je namijenjena. Kako bismo mrežu naučili da taj posao bolje obavlja moramo minimizirati funkciju gubitka. Iz matematičke analize je poznato da gradijent funkcije gubitka u nekoj točki s obzirom na proizvoljnu težinu ili prag nekog neurona pokazuje smjer najbržeg rasta funkcije gubitka, dok negativni gradijent pokazuje smjer najbržeg pada. Ideja gradijentnog spusta jest da ako želimo minimizirati funkciju mijenjamo težine u smjeru negativnog gradijenta. Dakle imamo:

$$\begin{aligned} w^{(k+1)} &= w^{(k)} - \lambda \Delta w^{(k)}, \\ \Delta w^{(k)} &= \frac{\partial \tilde{L}}{\partial w^{(k)}}, \end{aligned} \tag{3.13}$$

gdje je w neka proizvoljna težina ili prag nekog neurona, Δw iznos za koji treba promijeniti težinu, a k označava broj iteracije. λ se naziva parametar (stopa) učenja (engl. *learning rate*) koji je slobodan hiperparametar koji mi određujemo i govori koliko daleko se želimo pomaknuti u smjeru negativnog gradijenta (ako se izabere prevelik može uzrokovati da ne možemo završiti u minimumu, a ako je premalen učenje može trajati vrlo dugo).

Ova metoda vodi na potencijalne probleme. Naime, moguće je da učenje završi u lokalnom minimumu funkcije gubitka umjesto u globalnom, ipak, ova metoda se pokazala vrlo korisna i njome se mogu postići dobri rezultati. Intuitivan odgovor zašto učenje ne završi u lokalnom minimumu funkcije gubitka možemo potražiti u velikoj dimenzionalnosti prostora u kojem se odvija potraga za minimumom. Da točka prostora bude lokalni minimum, nužno je da parcijalna derivacija po svakoj od koordinatnih osi (njih n gdje n dimenzija prostora, odnosno broj slobodnih parametara) iščezava, dok za sedlo, samo neke od derivacija moraju iščezavati pa možemo zamisliti kako će se u takvom prostoru češće pojavljivati sedla (jer ih je lakše ostvariti),

nego lokalni minimumi.



Slika 3.6: Usporedba stohastičkog i običnog gradijentnog spusta. Posuđeno iz [11].

Sedla se, naime, mogu zaobilaziti koristeći stohastički gradijentni spust (odnosno engl. *online* učenje), tj. ažuriranje težina nakon što je kroz mrežu provučen samo jedan (ili manji broj) ulaznih podataka. Time se ne krećemo direktno prema minimumu nego ”kružimo” kao što se može vidjeti na slici 3.6, što pogoduje lagano, izvlačenju iz sedla.

Iako je dana jednadžba 3.13, nije opisano na koji način algoritam propagacije unatrag efikasno iskorištava pravilo ulančavanja derivacija. Izračunajmo prvo pravilo za ažuriranje težina neurona u zadnjem sloju. Neka je dana funkcija gubitka $\tilde{L}$ te mreža opisana funkcijom F kao u jednadžbi 3.7. Označimo težinu u zadnjem sloju neuronske mreže s $w_{ij}^{(m)}$ gdje m označava indeks zadnjeg sloja, indeks j odgovara j-toj komponenti ulaza u zadnji sloj ($h_j^{(m-1)}$), a indeks i odgovara i-toj komponenti izlaza ($\hat{y}_i$), dakle težina $w_{ij}^{(m)}$ jest težina koja određuje doprinos j-tog ulaza u zadnji sloj i-tome izlazu iz zadnjeg sloja. Označimo još s $\boldsymbol{o}^{(i)} = W_i\boldsymbol{h_{i-1}} + \boldsymbol{b_i}$ te s $f_j^{(i)} = f(o_j^{(i)})$ (što su komponente vektora $\boldsymbol{f}^{(i)}$) gdje je i indeks sloja neuronske mreže, a j, j-ta komponenta vektora $\boldsymbol{o}^{(i)}$. Za zadnji sloj vrijedi $f^{(m)} = \hat{\boldsymbol{y}}$, a za unutarnje $f^{(i)} = \boldsymbol{h}_i$ Tada imamo:

$$\Delta w_{ij}^{(m)} = \frac{\partial \tilde{L}}{\partial w_{ij}^{(m)}} = \frac{\partial \tilde{L}}{\partial f_i^{(m)}} \frac{\partial f_i^{(m)}}{\partial o_i^{(m)}} \frac{\partial o_i^{(m)}}{\partial w_{ij}^{(m)}}, \tag{3.14}$$

a kako općenito vrijedi

$$\frac{\partial o_i^{(k)}}{\partial w_{ij}^{(k)}} = \frac{\partial}{\partial w_{ij}^{(k)}}\Big(\sum_l w_{il}^{(k)} f_l^{(k-1)} + b_i^{(k)}\Big) = f_j^{(k-1)}, \tag{3.15}$$

imamo:

$$\Delta w_{ij}^{(m)} = \frac{\partial \tilde{L}}{\partial f_i^{(m)}} \frac{\partial f_i^{(m)}}{\partial o_i^{(m)}} f_j^{(m-1)} = \tilde{L}'(f_i^{(m)}) f'(o_i^{(m)}) f_j^{(m-1)} \tag{3.16}$$

gdje je prva parcijalna derivacija po redu samo derivacija funkcije gubitka $\tilde{L}'$, a druga je derivacija aktivacijske funkcije f' te su obje te derivacije poznate.

Ako sada izaberemo težinu $w_{ij}^{(k)}$ u nekom proizvoljnom sloju k, koji nije zadnji sloj, tada izvod više nije toliko jednostavan. Tada derivacija "ne vidi" (za nju su konstantne) sve težine iz ranijih slojeva. Ako primjetimo da samo neuron iz sloja $f_i^{(k)}$ ovisi o $w_{ij}^{(k)}$, tada imamo:

$$\Delta w_{ij}^{(k)} = \frac{\partial \tilde{L}}{\partial f_i^{(k)}} \frac{\partial f_i^{(k)}}{\partial w_{ij}^{(k)}} = \frac{\partial \tilde{L}}{\partial f_i^{(k)}} f'(o_i^{(k)}) f_j^{(k-1)}, \tag{3.17}$$

Nadalje računamo $\frac{\partial \tilde{L}}{\partial f_i^{(k)}}$ gdje $f_i^{(k)}$ ulazi u sve neurone sljedećeg sloja:

$$\frac{\partial \tilde{L}}{\partial f_i^{(k)}} = \sum_{l=1}^{N_{k+1}} \frac{\partial \tilde{L}}{\partial f_l^{(k+1)}} \frac{\partial f_l^{(k+1)}}{\partial f_i^{(k)}}. \tag{3.18}$$

Opet imamo:

$$\frac{\partial f_l^{(k+1)}}{\partial f_i^{(k)}} = f'(o_l^{k+1}) \frac{\partial o_l^{k+1}}{\partial f_i^{(k)}} = f'(o_l^{k+1}) w_{li}^{(k+1)} \tag{3.19}$$

pa jednadžba 3.18 postaje:

$$\frac{\partial \tilde{L}}{\partial f_i^{(k)}} = \sum_{l=1}^{N_{k+1}} \frac{\partial \tilde{L}}{\partial f_l^{(k+1)}} f'(o_l^{k+1}) w_{li}^{(k+1)}. \tag{3.20}$$

Ako sada uvedemo oznaku, koristeći 3.16 3.17 i 3.20:

$$\delta_i^{(k)} = \frac{\partial \tilde{L}}{\partial f_i^{(k)}} \frac{\partial f_i^{(k)}}{\partial o_i^{(k)}} = f'(o_i^{(k)}) \times \begin{cases} \sum_{l=1}^{N_{k+1}} w_{li}^{(k+1)} \delta_l^{(k+1)} & unutarnji\ neuron \\ \tilde{L}'(f_i^{(k)}) & izlazni\ neuron, \end{cases} \tag{3.21}$$

koja predstavlja rekurziju koja se zaustavlja tek u zadnjem sloju neuronske mreže, možemo pisati:

$$\Delta w_{ij}^{(k)} = \delta_i^{(k)} f_j^{(k-1)}, \tag{3.22}$$

gdje je $\delta_i^{(k)}$ određen s rekurzivnom jednadžbom 3.21.

Iz rekurzivne jednadžbe 3.21 možemo uvidjeti sljedeće: računanjem $\delta_i^{(m)}$ za sve

neurone u zadnjem sloju, već smo izračunali velik dio gradijenta za neurone u predzadnjem sloju i to se analogno nastavlja do prvog sloja mreže. Dakle izračunati gradijenti se propagiraju nazad kroz mrežu i služe u računanju gradijenata koji su dublje u mreži. Otuda dolazi i sam naziv propagacija greške unazad. Analogan račun se može napraviti i za pragove, ili se može pojednostaviti time da se uvede još jedna dimenzija kod ulaza u neki neuron čija vrijednost uvijek iznosi 1 pa se onda prag može gledati kao $(n+1)$. težina i samim time se ovaj račun može iskoristiti za računanje ažuriranja praga.

3.6 Teorem univerzalne aproksimacije

Teorem univerzalne aproksimacije govori [12] o tome da unaprijedna (engl. feedforward) neuronska mreža s jednim skrivenim slojem i konačnim brojem neurona može proizvoljno dobro opisati neku kontinuiranu funkciju f na kompaktnom potprostoru od $\mathcal{R}^n$ uz blage pretpostavke za aktivacijsku funkciju. Sada možemo formalno iskazati teorem.

Neka je ϕ: $\mathcal{R} \to \mathcal{R}$ nekonstantna, ograničena te kontinuirana funkcija. Neka je I_m m-dimenzionalna jedinična hiperkocka $[0,\ 1]^m$. Označimo s $C(I_m)$ prostor realnih funkcija na I_m. Onda, za proizvoljan ϵ i proizvoljnu funkciju $f \in C(I_m)$ postoji prirodan broj N, realne konstante v_i, b_i i realni vektori $w_i \in \mathcal{R}^m$ za svaki $i = 1,\ 2,\ ...,\ N$ takvi da možemo definirati:

$$F(x) = \sum_{i=1}^{N} v_i \phi(w_i^T x + b_i), \tag{3.23}$$

takvu da vrijedi:

$$|F(x) - f(x)| < \epsilon,\ \forall x \in I_m. \tag{3.24}$$

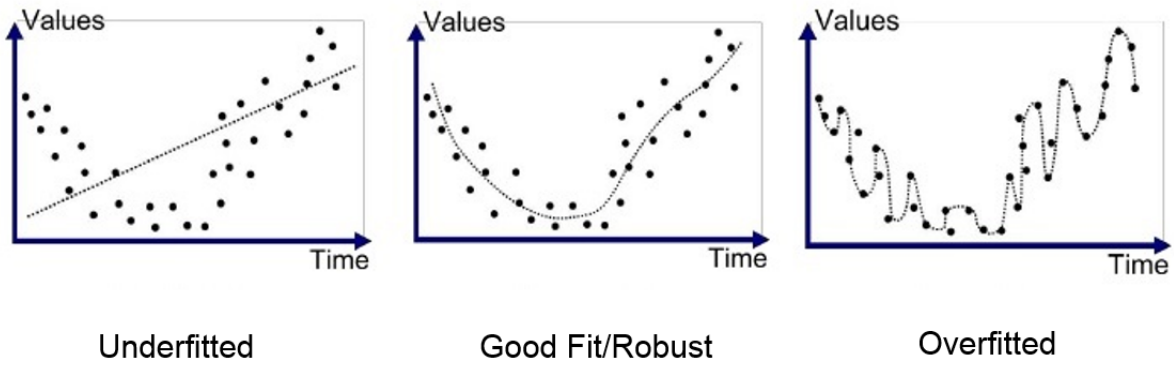
Drugim riječima, funkcije oblika $F(x)$ su guste u $C(I_m)$.

Iako teorem osigurava da neuronske mreže mogu opisati proizvoljnu funkciju, nemamo garanciju da ćemo mi tu funkciju pronaći. Naime, algoritam može završiti u lokalnom minimumu umjesto u globalnom. Nadalje, podaci u stvarnom svijetu su šumoviti. Samim time, učenjem ne želimo naučiti direktno podatke koje vidimo, već funkciju koja ih je generirala prije nego što je na njih djelovao šum. Pristup tome problemu bit će opisan u sljedećem poglavlju.

3.7 Podjela skupa podataka

Kako neuronskom mrežom ne bismo učili šum, dijelimo cijeli skup ulaznih podataka $D = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}$ za $i = 1,\ 2,\ ...,\ N$ gdje je N ukupan broj podataka, na skup

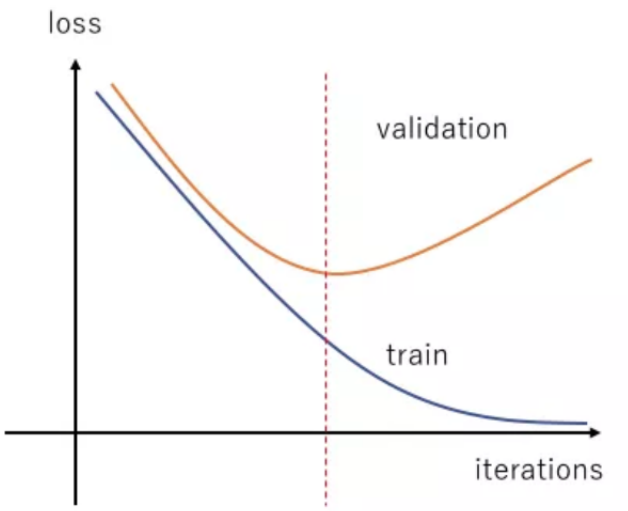
za treniranje i skup za validaciju u nekom proizvoljnom omjeru (npr. $2:1$). Mrežu učimo na skupu za treniranje te nakon svake epohe (jednog prolaska na cijeli skup za treniranje) gledamo pogrešku na skupu za validaciju.



Slika 3.7: Prikaz podnaučenosti, željenog rezultata te prenaučenosti. Posuđeno s [13].

U grubo razlikujemo dvije situacije. U prvoj situaciji pogreška na skupu za treniranje i za validaciju pada s brojem iteracija i koliko god trenirali obje greške ne počinju rasti. To je slučaj kada nemamo model (neuronsku mrežu) dovoljnog kapaciteta (model nije dovoljno složen, odnosno ekspresivan) kao što je, na primjer, prikazano na lijevoj strani slike 3.7. Ova situacija se zove podnaučenost.

Druga situacija je kada pogreška na skupu za treniranje pada, dok pogreška na skupu za validaciju pada te u jednom trenutku počinje rasti. U ovoj situaciji, model je dovoljnog kapaciteta te u tom trenutku prestaje učiti funkciju koju želimo naučiti (funkciju koja je generirala podatke bez šuma) te počinje učiti šum jer je dovoljno ekspresivna da ga opiše. Ovaj scenarij je prikazan na desnoj strani slike 3.7.



Slika 3.8: Prikaz pogreške na skupu za treniranje i validaciju kod dovoljno ekspresivnog modela. Posuđeno s [14].

Ovaj problem možemo riješiti tako da mijenjamo model sve dok ne dobijemo model koji je otprilike između ove dvije situacije. Drugi pristup (često i brži jer ne moramo mnogo puta trenirati mrežu već samo jednom) je taj da uzmemo model velikog kapaciteta te prestanemo s treningom u trenutku kada greška na skupu za validaciju počne rasti. Takva situacija je prikazana na slici 3.8.

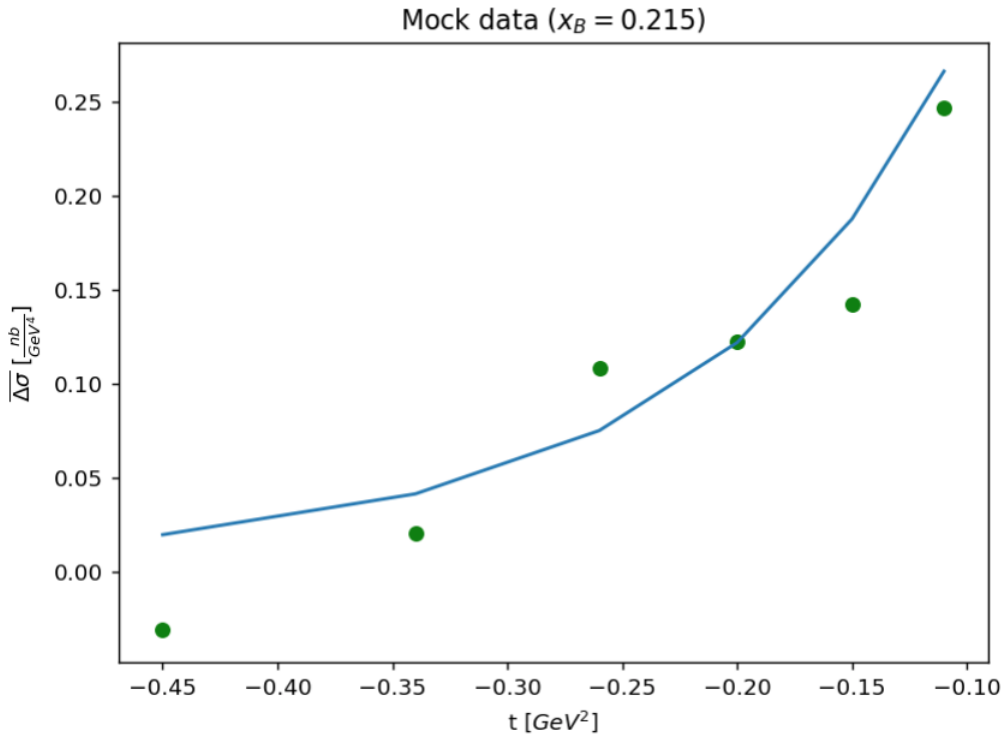
Također, prenaučenost možemo spriječiti koristeći tzv. regularizaciju. To je postupak u kojem funkciji gubitka dodajemo još jedan član koji kažnjava složenost modela, jer složeniji modeli su skloni prenaučenosti. To se na primjer može napraviti na način da se funkciji gubitka doda suma normi težina (gdje norma može biti *L1*, *L2* (vidi [15]), ...). Dodana mjera složenosti uključuje se u funkciju gubitka proporcionalno s konstantom regularizacije koja predstavlja još jedan hiperparametar.

Važno je i napomenuti da su i podaci dobar regularizator tj. da puno različitih podataka također može spriječiti prenaučenost.

4 Prilagodba neuronskih mreža na umjetne podatke

U ovome radu želimo testirati koliko dobro možemo naučiti komptonske form faktore (CFF-ove) modelirane neuronskim mrežama, iz tipičnih eksperimentalnih podatka. Važno je uočiti da ovdje ne učimo direktno CFF funkcije već učimo ovisnost ulaznih varijabli o opservablama mjerenim u eksperimentu koje ovise o CFF funkcijama. Dakle, CFF funkcije su modelirane neuronskim mrežama (zasebno njihov realni i imaginarni dio, dakle dvije neuronske mreže za jednu CFF funkciju) te se njihovi izlazi kombiniraju u opservable.

Kako CFF funkcije još nisu precizno poznate, a želimo odrediti koliko dobro je moguće ekstrahirati CFF funkcije iz tipičnog eksperimentalnog postava, odlučili smo simulirati eksperiment (generirati umjetne podatke kao na slikama 4.1 i 4.2) te na njima se uvjeriti koliko dobro možemo ekstrahirati CFF funkcije.

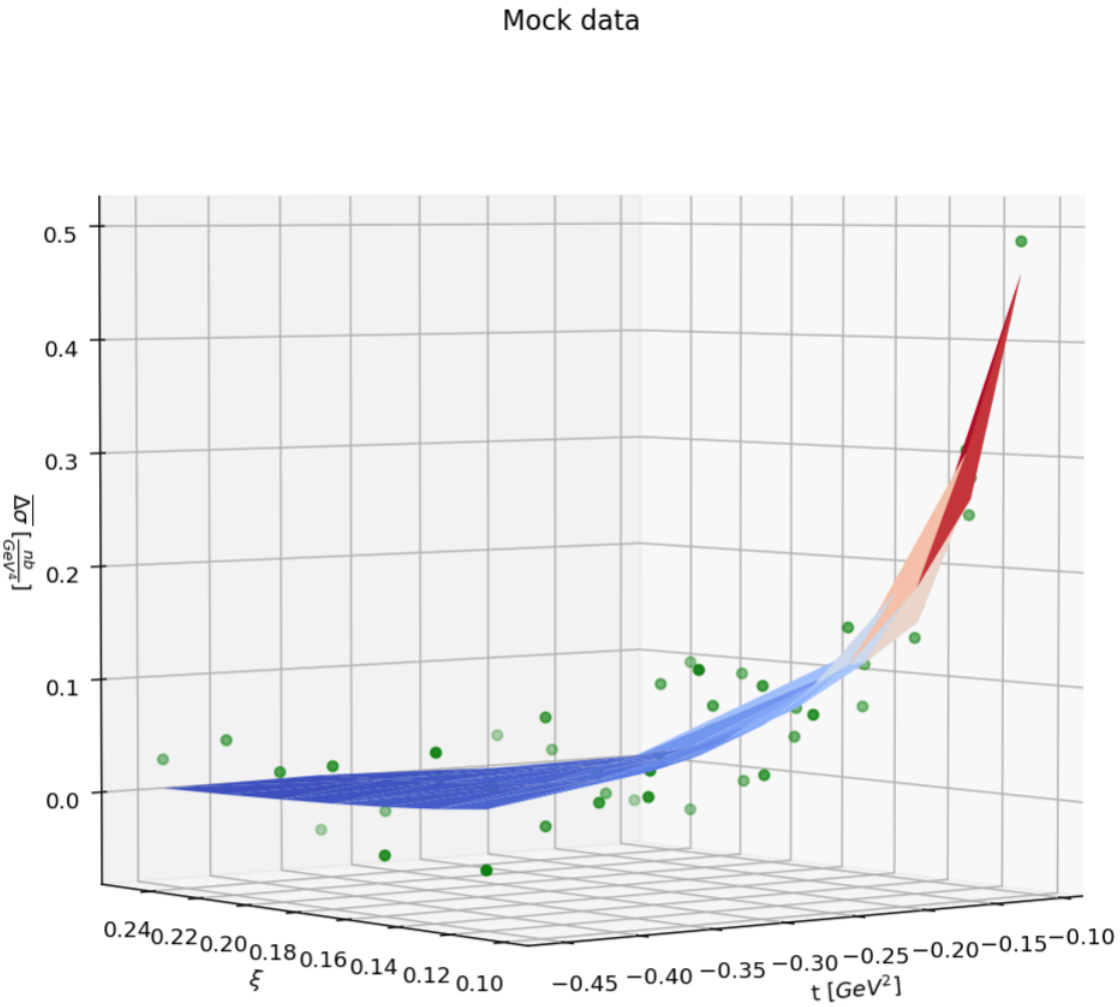


Slika 4.1: Prikaz umjetnih podataka za opservablu $\overline{\Delta\sigma}$ (engl. *weighted beam spin difference*) uz fiksan $x_B = 0.215$.

4.1 Umjetni podaci

U ovom je potpoglavlju opisan postupak generiranja umjetnih (engl. *mock*) podataka. Kako bi se vjerno simulirao eksperiment, za vrijednosti ulaznih varijabli t i ξ, o kojima ovise CFF funkcije, odnosno opservable, uzete su vrijednosti iz eksperimenata na CLAS detektoru [17]. U svakom eksperimentu imamo situaciju da se prave CFF

funkcije, koje ne znamo, za određene t i ξ evaluiraju te se kombiniraju u opservable. Zbog nesavršenosti mjerenja u realnim eksperimentima uvijek postoji određeni šum te je ono što mjerimo u eksperimentu zapravo superpozicija opservable i šuma.



Slika 4.2: Prikaz umjetnih podataka za opservablu $\overline{\Delta\sigma}$ (engl. *weighted beam spin difference*).

Taj postupak sada želimo modelirati. Prvo je potreban što vjerniji model CFF funkcija za koji uzimamo poznati Goloskokov-Kroll (GK) model [18]. Pomoću GK modela CFF funkcija generiramo različite opservable na koje dodajemo gausijanski šum. Na slici 4.1 je dan prikaz opservable $\overline{\Delta\sigma}$ (engl. *weighted beam spin difference*), uz fikasn x_B, modelirane u GK modelu te umjetni podaci na kojim učimo neuronske mreže, dok je na slici 4.2 dana ista opservabla za različite t i ξ. Primjer kôda za generiranje umjetnih podataka dan je u dodatku C.

4.2 Prilagodba postojećeg kôda na TensorFlow

Neuronske mreže modelirane su u *TensorFlow* (*tf*) okruženju. *TensorFlow* je *python*-ova biblioteka (engl. *open source library*) razvijena od strane Google-a na području umjetne inteligencije (*artificial intelligence*, AI) i danas je jedan od najkorištenijih AI sustava danas. Dobiven je pristup mentorovom kôdu te je bilo potrebno prilagoditi ga na *TensorFlow*. Primjeri kôda dani su u dodacima.

TensorFlow je koncipiran na drugačiji način od uobičajenog kôda. Naime, obične operacije (npr. zbrajanje dvije varijable) se pri izvršavanju *TensorFlow* kôda ne računaju odmah kao što je to uobičajeno. U *TensorFlow*-u se prvo gradi tzv. graf koji predstavlja npr. neuronsku mrežu, a samo računanje se događa unutar tzv. sjednice (engl. *tf.Session*).

Kôd 1: Primjer jednoslojne neuronske mreže napisane u *TensorFlow*-u

```
1  import tensorflow as tf
2
3  x = tf.placeholder(...)
4  y = tf.placeholder(...)
5  w = tf.Variable(...)
6  v = tf.Variable(...)
7  b = tf.Variable(...)
8
9  y_hat = v*tf.sigmoid(tf.matmul(w,x)+b)
10
11 loss = tf.square(y-y_hat)
12
13 train_op = tf.train.AdamOptimizer().minimize()
```

Dakle, ako se želi definirati neuronsku mrežu, prvo definiramo ulaze u neuronsku mrežu (*tf.placeholder*), potom težine i pragove za svaki sloj (*tf.Variable*), te se kao u običnom programiranju definiraju operacije koje računaju izlaz iz mreže na temlju ulaza, težina i pragova. Također postoje već definirani razni slojevi neuronskih mreža koji se mogu korisiti (npr. postoji potpuno povezani sloj - tf.layers.dense). Nadalje definiraju se i željeni izlazi tj. varijable koje predstavljaju ono što očekujemo na izlazu iz neuronske mreže, opet u obliku *tf.placeholder*-a. Koristeći izlaz iz neuronske mreže te varijablu očekivanog izlaza možemo definirati funkciju gubitka.

Sada možemo odabrati jedan od već implementiranih algoritama za učenje neuronske mreže (u ovom radu je korišten Adam algoritam) te se on primjeni na funkciju gubitka i daje čvor u grafu koji predstavlja operaciju učenja (jedan korak treniranja) mreže. Pojednostavljeni primjer izgradnje jednoslojne neuronske mreže dan je u kôdu 1 (argumenti *tf.placeholder* i *tf.Variable* varijabli su odgovarajući oblik i tip podatka kojeg predstavljaju).

Kôd 2: Primjer učenja neuronske mreže u *TensorFlow*-u

```
...
data_x = ...
data_y = ...
num_epochs = 100

with tf.Session() as sess:
        for i in range(num_epochs):
                for x_in, y_in in zip(data_x, data_y):
                        sess.run(train_op, feed_dict={x: x_in, y:
                            y_in})
```

Kôd napisan do sada nije se izvršio, njegov rezultat je stvaranje grafa koji predstavlja neuronsku mrežu, funkciju gubitka, te operaciju treniranja (računanja i primjene gradijenata težina). Mrežu se sada trenira unutar već spomenute sjednice (*tf.Session*) tako da predamo mreži različite podatke (ulaz i očekivani izlaz), ulazi prolaze unaprijed kroz graf te se računaju izlazi iz neuronske mreže koji se koriste pri računanju funkcije gubitka. Na temelju funkcije gubitka računaju se gradijenti koji se propagiraju unatrag kroz graf (propagacije greške unazad) te se ažuriraju težine mreže. *TensorFlow* je koristan zbog toga što se diferenciranje (računanje gradijenata) svakog čvora u grafu odvija automatski. Pojednostavljeni primjer treniranja mreže dan je u kôdu 2, gdje se funkcijom *sess.run* izvršavaju svi čvorovi u grafu o kojima ovisi *train_op* čvor. Također, vrijedno je primjetiti da smo morali specificirati sve *tf.placeholder* varijable kako bi se kôd mogao izvršiti.

Kako bi se postojeći kôd za računanje opservabli iz danih CFF funkcija mogao izvršavati u *TensorFlow*-u bilo je potrebno napraviti određene promjene. Na početku su se definirale neuronske mreže za korištene CFF funkcije, tj. stvoren je njihov dio grafa što se može vidjeti kod definicije klase *ToyModel* u dodatku B. Sada je bila ideja iskoristiti postojeći kôd za računanje različitih opservabli iz izlaza CFF funkcija. To je napravljeno na način da su se sve funkcije iz *numpy* (*np*) biblioteke korištene u postojećem kôdu (npr. np.sin, np.cos, np.log, ...) morale zamijeniti s *TensorFlow* funkcijama koje grade graf (tf.sin, tf.cos, tf.log, ...). Nadalje, svi *if - then - else* blokovi su se morali zamijeniti s analognom *TensorFlow*-ovom *tf.cond* funkcijom koja se ne izvršava u trenutku prevođenja kôda (kao *if* blok) već u odgovarajućoj sjednici.

Postojeći kôd za računanje opsevabli iz CFF funkcija kao ulaz prima klasu DataPoint čiji atributi su zamijenjeni s *tf.placeholder*-ima. No, onda na temelju predate instance DataPoint klase stvara novu instancu DataPoint te joj računa atribute čime se izgube predani *tf.placeholder*-i. To se riješilo na način da klasa koja sadrži CFF funkcije, pri inicijalizaciji prima i zapamti instancu DataPoint klase koja kao atribute ima *tf.placeholder*-e. Onda pri pozivu izračunavanja CFF funkcije, zanemaruje

primljeni argument (novostvorenu instancu DataPoint klase s prebrisanim određenim atributima) i kao ulaz koristi zapamćenu instancu DataPoint klase. Konkretna implementacija se može vidjeti u dodatku B.

Nadalje se postojeći kôd trenira te se nakon svake epohe (jednog prolaza kroz sve podatke) izvršava validacija (određivanje koliko dobro model radi na skupu za validaciju) te ako je model bolji no što je bio pri prošloj validaciji, spremamo težine modela (ovdje koristimo koristan alat *checkmate* koji pamti samo određen broj zadnjih najboljih instanci modela). To možemo vidjeti u dodatku C.

Kompletan primjer učenja mreže dan je obliku *Jupyter* bilježnice u dodatku D.

4.3 Rezultati

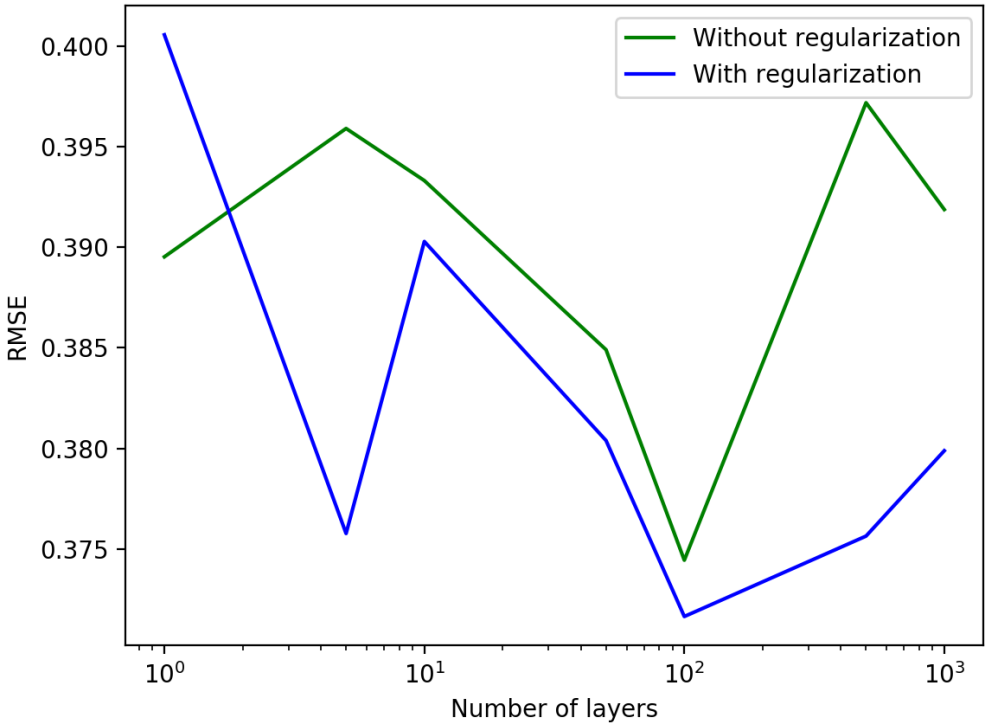
U ovom potpoglavlju dan je pregled provedenih mjerenja te rezultate koji su dobiveni. Na početku moraju se odrediti hiperparametri (broj slojeva u mreži, broje neurona u svakom sloju, ...) mreže. U svim učenjima mreža koristili smo podjelu skupa podataka na skup za treniranje te skup za validaciju u omjeru 9:1, za funkciju pogreške korišteno je kvadratno odstupanje, za učenje neuronske mreže korišten je poznati Adam algoritam (od engl. *Adaptive momentum* opisan u [19]), korišteno je *online* učenje (nakon viđenog jednog primjera bi se ažurirale težine i pragovi), kao funkcija gubitka korištena je suma kvadrata odstupanja te kao metrika točnosti regresije između naučenih i pravih CFF funkcija korištena je greška korijena iz srednjeg kvadratnog odstupanja (engl. *root mean square error* (RMSE)), koja je definirana između dva skupa točnih ((y_i) za $i = 1,\ 2,\ ...,\ N$) i naučenih ($(\hat{y}_i)$ za $i = 1,\ 2,\ ...,\ N$) vrijednosti kao:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}}. \tag{4.1}$$

Zbog složenosti modela nismo mogli iskoristiti benefite treniranja neuronskih mreža na grafičkim karticama (GPU-ovima) te tako ubrzati eksperimente. Naime, trening na GPU-u je čak nekoliko puta bio sporiji nego trening na procesoru (CPU-u), čemu je vjerojatno razlog to da nakon izračunavanja CFF funkcija slijedi velik dio kôda za računanje opservabli iz CFF funkcija koji se, pretpostavljma, nije mogao previše ubrzati.

Kako različite podjele podataka ne moraju voditi do jednakih rezultata (jednako naučene mreže), a i učenje modela (pretraživanje prostora težina i pragova) ne mora završiti u istom stanju za uzastopna pokretanja, odlučeno je pokrenuti iste eksperimente više puta pa uzeti njihov prosjek. Zbog velike duljine trajanja svakog eksperimenta, zbog gore opisanih problema, odlučili smo pokrenuti isti eksperiment dva puta, svaki put s različitom podjelom na skup za treniranje i validaciju te promatrati

njihov prosjek.



Slika 4.3: Prikaz ovisnosti RMSE o broj slojeva, uz fiksan broj neurona po mreži jednak 100.

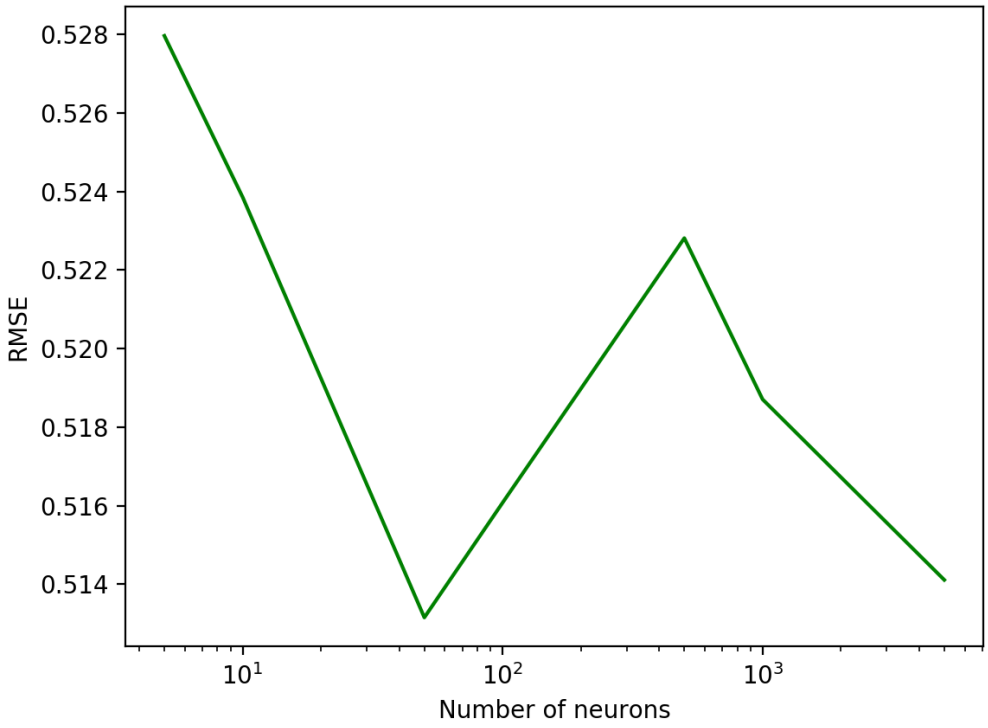
Kako bi se što bolje odredio broj slojeva u mreži provedeni su eksperimenti s različitim brojem slojeva u mreži, za fiksan (jednak 100) broj neurona u svakom sloju. Provedeni su eksperimenti za 1, 5, 10, 50, 100, 500 i 1000 slojeva te su dobiveni rezultati prikazani na slici 4.3. Na slici je dana ovisnost RMSE o broju slojeva u mreži za dva slučaja: zelena linija prikazuje ovisnost RMSE o broju slojeva u mreži bez uključene regularizacije, dok plava linija prikazuje slučaj s uključenom regularizacijom uz *L2* normu. Iz rezultata eksperimenata bi se mogla očitati RMSE za svaki pojedini proučavani CFF (proučavali su se samo imaginarni dijelovi $\mathcal{H}$ i $\tilde{\mathcal{H}}$ te realni dio H CFF funkcije, označeni redom *ImH*, *ImHt* te *ReH*, iz razloga što oni najviše doprinose proučavanim opservablama kao što je opisano u [16]) te je računata aritmetička sredina između njih. Dakle jedna linija predstavlja srednju vrijednost (između eksperimenata) aritmetičkih sredina između RMSE-ova za svaki proučavani CFF, ili konzistentnije opisano formulom:

$$\overline{RMSE} = \frac{\sum_{i=1}^{N_{exp}} (RMSE^i_{ImH} + RMSE^i_{ReH} + RMSE^i_{ImHt})/3}{N_{exp}}, \tag{4.2}$$

gdje $RMSE_{cff}$ predstavlja dobivenu grešku RMSE za cff CFF funkciju, dok i označava indeks, a $N_{exp} = 2$ ukupan broj provedenih eksperimenata s istim postavom.

Sa slike možemo vidjeti da modeli s regularizacijom bolje rade za kompleksnije modele što je bilo za očekivati. Kako se minimum RMSE, u oba slučaja, postiže za 100 slojeva, odlučeno je raditi s mrežom od 100 slojeva te od sada, ako nije drugačije navedeno, podrazumijeva se da mreža ima 100 slojeva. Također, kako je mreža sa 100 slojeva kompleksna, u svim sljedećim eksperimentima koristit ćemo *L2* regularizaciju. Iako je za 100 slojeva dobiven najbolji rezultat, iz grafa se nije jasno je li zaista potrebno toliko slojeva mreže (vidi se da se i s 5 slojeva dobiva samo nešto lošiji rezultat), ali se jasno pokazuje utjecaj regularizacije u dubokim mrežama.

Nadalje, kako bi se odabrao optimalan broj neurona, proveli smo eksperimente s različitim brojem neurona. Na slici 4.4 prikazana je ovisnost RMSE (srednje RMSE računate kao gore) o broju neurona.

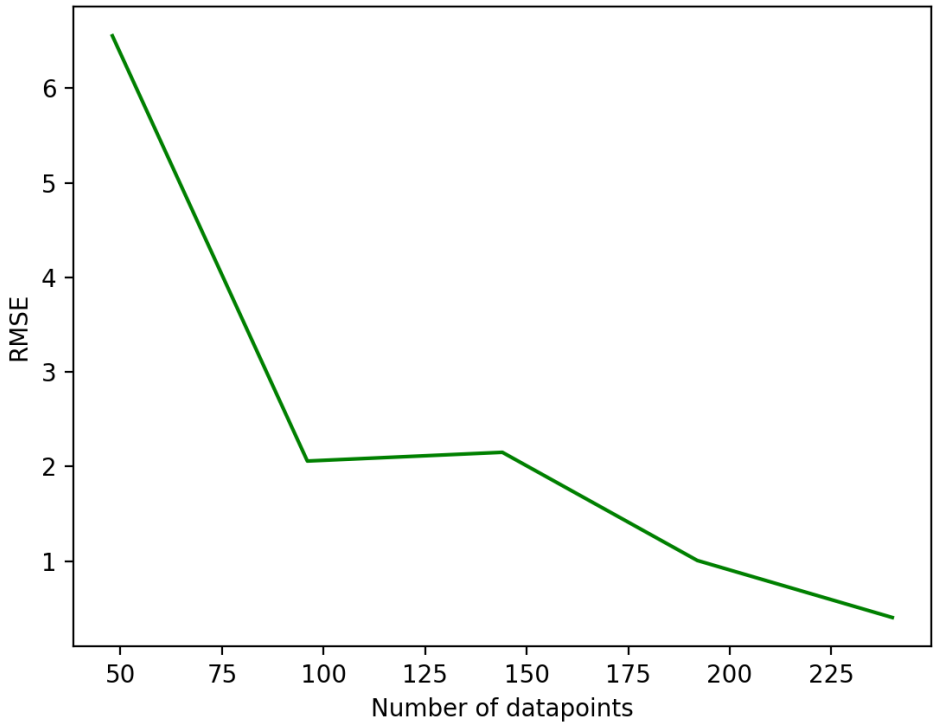


Slika 4.4: Prikaz ovisnosti RMSE o broj neurona, uz fiksan broj slojeva po mreži jednak 100.

Računata je RMSE za brojeve neurona: 5, 10, 50, 100, 500, 1000 i 5000. Broj neurona nije dalje povećavan zbog hardverskih ograničenja te zbog duljine treninga. Kako je za 50 neurona dobiven minimum srednjeg RMSE-a, taj broj neurona se koristi u ostatku rada. Moglo se i za ove eksperimente raditi pretraživanje za optimalnu

konstantu regularizacije (konstanta), no opet, zbog toga što bi to izrazito povećalo broj treninga, za nju je odabrana samo jedna vrijednost jednaka 0.05. Iz grafa se može vidjeti da RMSE za različite brojeve neurona ne varira puno, rezultat bi se mogao objasniti i slučajnim fluktuacijama između eksperimenata. Dakle, ne može se povući jasan zaključak je li nužno imati toliko puno neurona u svakom sloju.

Kako bi se odredio utjecaj broja točaka iz kojih mreža uči na RMSE CFF funkcija provedeni su eksperimenti s različitim podskupima ukupnog skupa podataka. Rezultati eksperimenata prikazani su na slici 4.5.



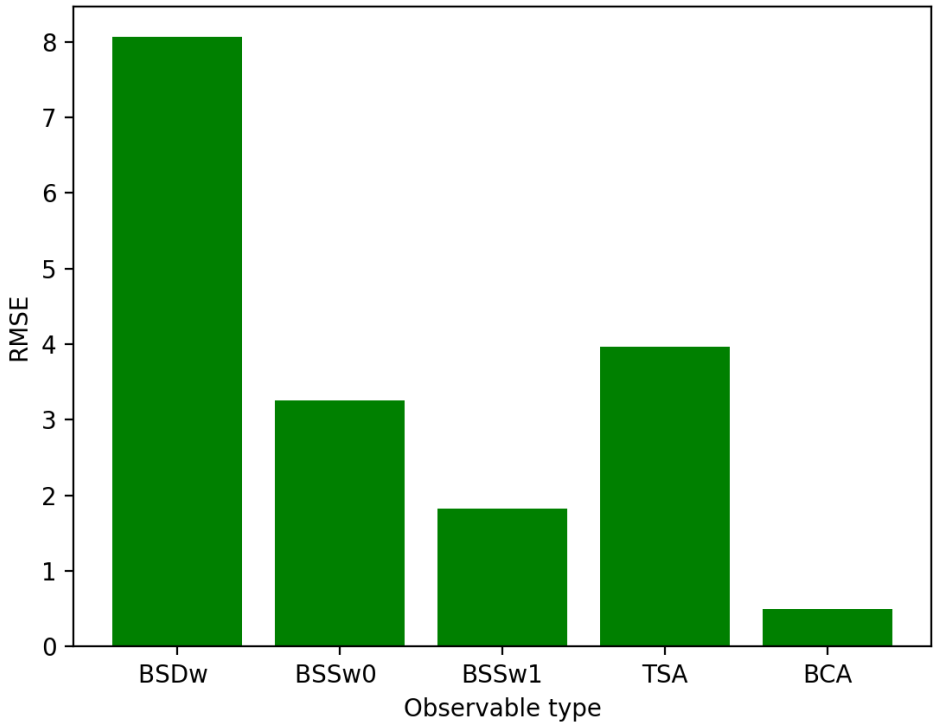
Slika 4.5: Prikaz ovisnosti RMSE o dostupnom broj podataka.

Točke na slici 4.5 prikazuju ovisnost o ukupnom broju podataka koji se onda dalje dijelio na skup za treniranje i skup za validaciju. Kao što je i očekivano, sa slike 4.5 se vidi trend pada RMSE s povećavanjem broja podataka. To je očekivano zbog toga što se uz više podataka mreža može bolje naučiti.

Nadalje, provedeni su eksperimenti ovisnosti RMSE o kombinaciji različitih opservabli. Prikazi rezultata eksperimenata podijeljeni su u grupe eksperimenata koji za podatke koriste 1, 2, 3, 4 ili svih 5 različitih opservabli. Korištene opservable na narednim slikama označene su kao $BSDw$, $BSSw0$, $BSSw1$, BCA i TSA, gdje BSD označava polarizirani udarni presjek (2.31), BSS nepolarizirani udarni presjek (2.30), BCA označava asimetriju naboja snopa (2.20), a TSA označava asimetriju

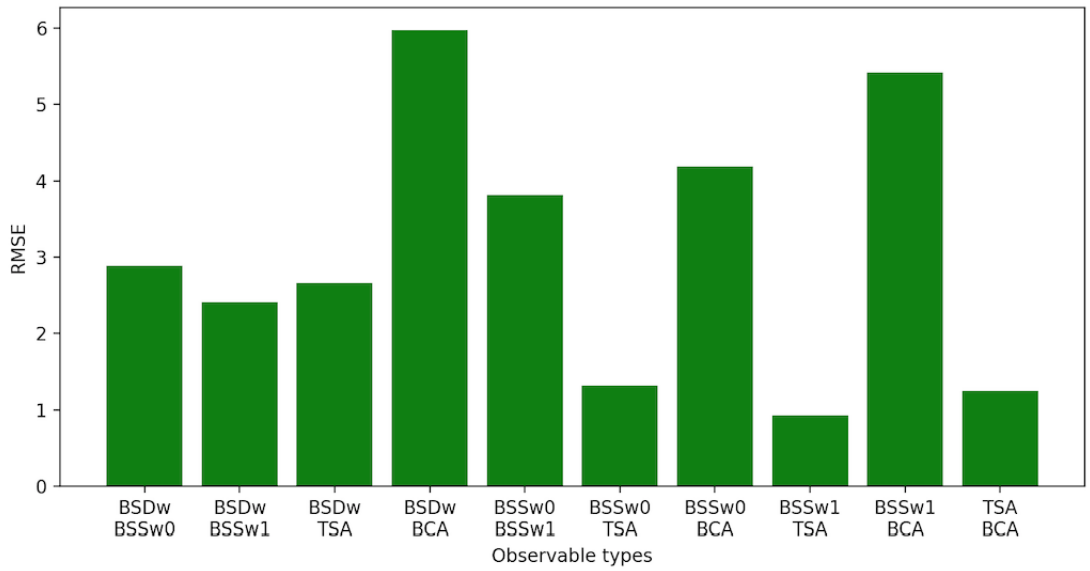
spina mete (2.24). Slovo w označava da se radi o vrsti varijable dobivene određenom vrstom harmonijske analize kao što je npr. opisano u jednadžbi (2.28). Ako uz opservablu stoji samo slovo w tada su težine dane kao $sin(\phi)$, a ako stoji i broj 0 ili 1 onda kao težine ne koristimo sinus već kosinus funkcije, tj. $cos(0\phi)$ (što je jednako 1 pa zapravo imamo samo srednju vrijednost te opservable po kutu ϕ) i $cos(1\phi)$.

Na slici 4.6 dan je prikaz ovisnosti RMSE o vrsti korištene opservable za slučaj kad se koristi samo jedna opservabla. Vidimo da se najbolji rezultati dobivaju korištenjem BCA opservable.



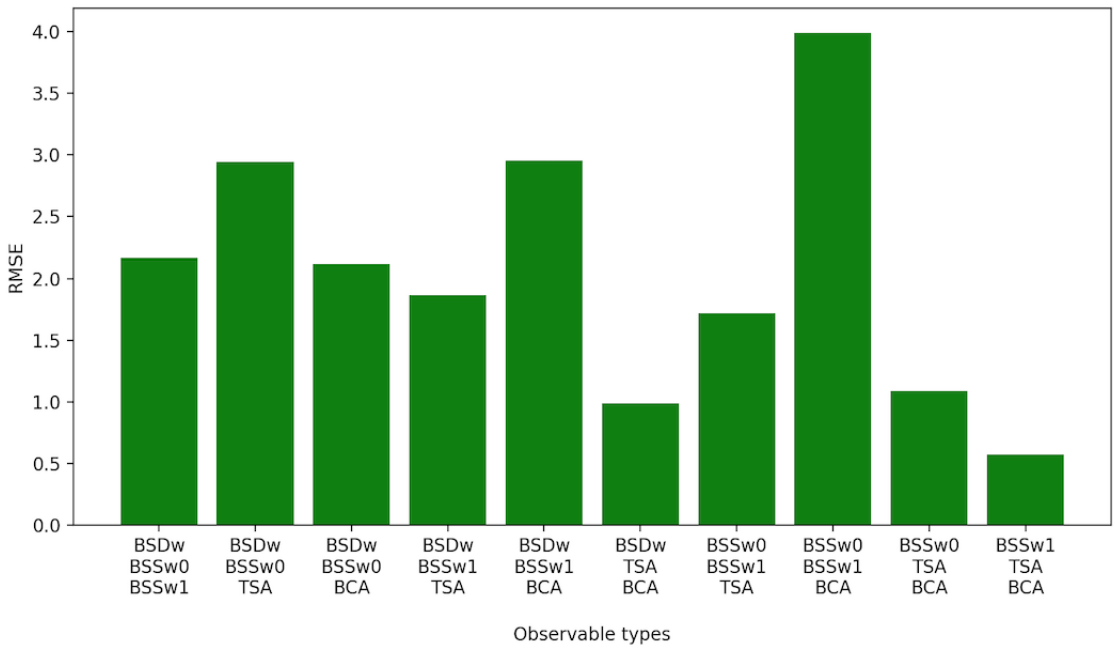
Slika 4.6: Prikaz ovisnosti RMSE o vrsti korištene opservable.

Na slici 4.7 dan je prikaz ovisnosti RMSE o vrsti korištene opservable za slučaj kad se koriste dvije opservable. Vidimo da se najbolji rezultati u ovom slučaju dobivaju korištenjem TSA opservable zajedno s BSS, odnosno BCA opservablom.



Slika 4.7: Prikaz ovisnosti RMSE o vrsti korištene opservable za slučaj dvije korištene opservable.

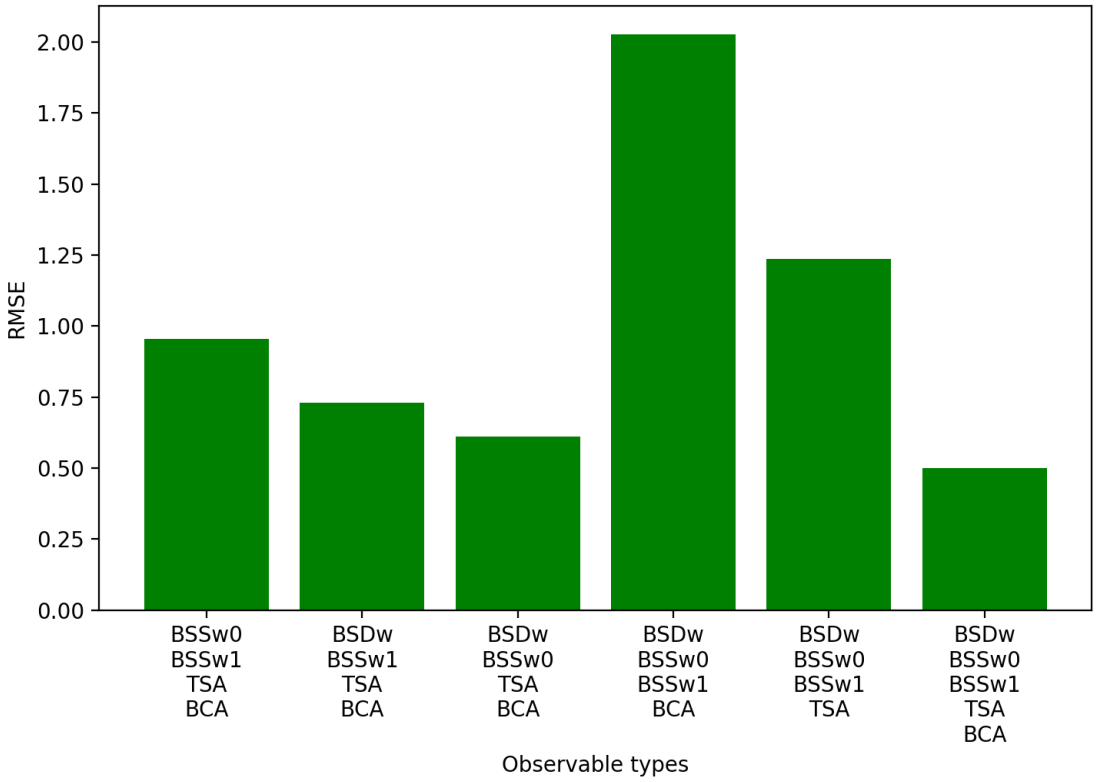
Na slici 4.8 dan je prikaz ovisnosti RMSE o vrsti korištene opservable za slučaj kad se koriste tri opservable. Vidimo da se najbolji rezultati u ovom slučaju dobivaju korištenjem TSA i BCA opservable zajedno s ostalim opservablama.



Slika 4.8: Prikaz ovisnosti RMSE o vrsti korištene opservable za slučaj tri korištene opservable.

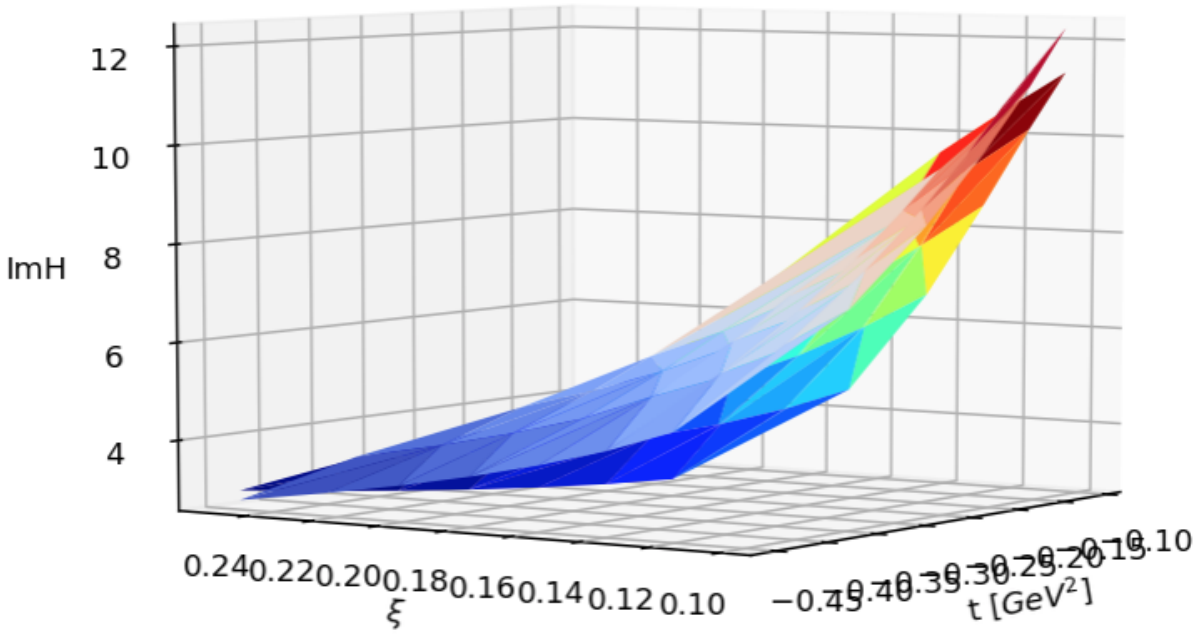
Nadalje, na slici 4.9 dan je prikaz rezultata zadnja dva eksperimenta za skupine od 4 odnosno 5 opservabli. Kao i u ranijim eksperimentima vidimo da korištenje

zajedno TSA i BCA opservable rezultira u boljem RMSE-u.

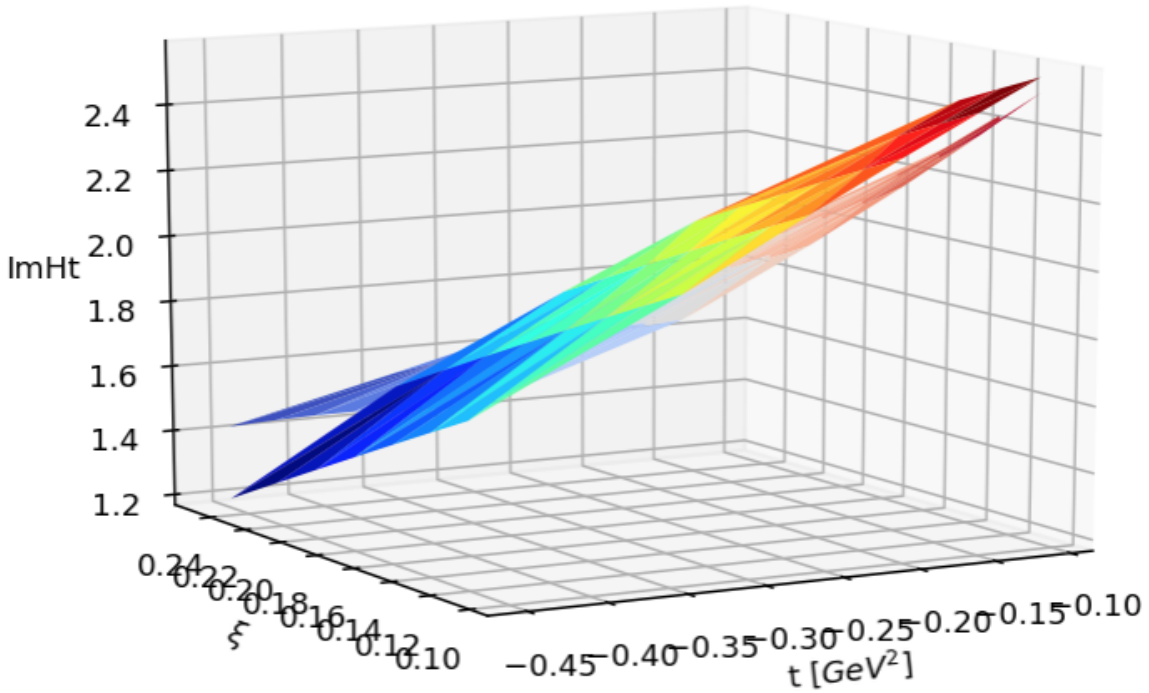


Slika 4.9: Prikaz ovisnosti RMSE o vrsti korištene opservable za slučaj četiri i pet korištenih opservabli.

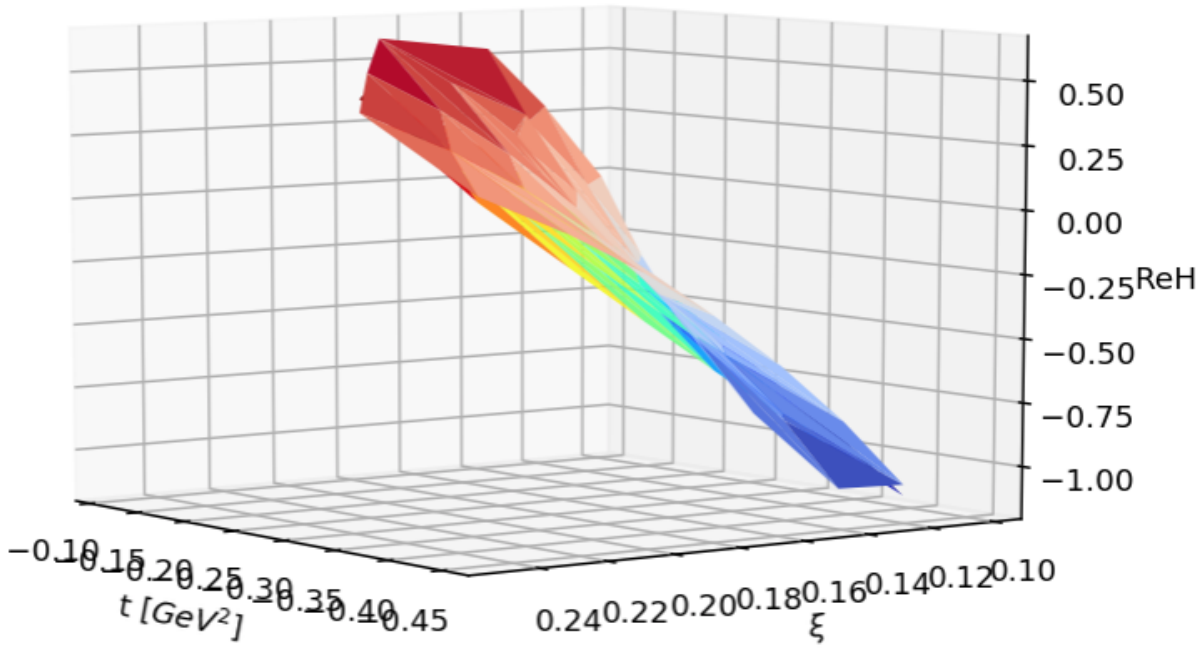
Nadalje na slikama 4.10, 4.11 i 4.12 dan je prikaz naučenih vodećih CFF funkcija na svim korištenim opservablama. Vidimo su relativno dobro naučene. Iz svih provedenih eksperimenata možemo dakle zaključiti da korištenje neuronskih mreža za ekstrakciju CFF funkcija iz izmjerenih opservabli može biti korisno. Nadalje, rezultati će biti to bolji što više podataka imamo te što više različitih opservabli koristimo. Važno je i uočiti da korištenjem TSA i BCA opservabli zajedno poboljšava ekstrakciju.



Slika 4.10: Prikaz naučene $Im\mathcal{H}$ CFF funckije. Gornja ploha predstavlja $Im\mathcal{H}$ u GK modelu, a donja predikciju.



Slika 4.11: Prikaz naučene $Im\tilde{\mathcal{H}}$ CFF funckije. Donja ploha predstavlja $Im\tilde{\mathcal{H}}$ u GK modelu, a gornja predikciju.



Slika 4.12: Prikaz naučene $Re\mathcal{H}$ CFF funckije. Gornja ploha predstavlja $Re\mathcal{H}$ u GK model, a donja predikciju.

5 Zaključak

U ovom radu proučena je mogućnost ekstrakcije CFF funkcija neuronskim mrežama iz opservabli koje se tipično mjere u eksperimentima. Opisane su GPD funkcije, čije bolje poznavanje će izaći iz boljeg poznavanja CFF funkcija. Opisane su i tipične opservable mjerene u procesima iz kojih želimo, između ostalog, odrediti CFF funkcije.

Nadalje, opisan je model kojim su modelirane CFF funkcije - neuronske mreže. Neuronske mreže su izabrane zbog toga što ne unose pristranost pri izboru modela, kao i poželjnog svojstva da su guste u prostoru realnih funkcija definiranih na jediničnoj hiperkocki tj. da su univerzalni aproksimatori. Dana je inspiracija za prvi model umjetnog neurona te je opisano kako se umjetni neuroni mogu kombinirati u umjetne neuronske mreže. Opisano je i na koji način se uči neuronska mreža.

Pokazano je na umjetnim podacima da je se neuronskim mrežama mogu ekstrahirati CFF iz tipično mjerenih opservabli, da će ekstrakcija biti to bolja što je više podataka te je uočena pravilnost da se korištenjem TSA i BCA opservable zajedno pri učenju neuronske mreže, dobivaju bolji rezultati.

Dodaci

Dodatak A mock_data.py

Ovdje možemo vidjeti primjer generiranja umjetnih podataka.

Kôd 3: *mock_data.py*

```
import os, sys, shelve, logging
from shutil import rmtree
from sklearn.model_selection import train_test_split
import numpy as np

# Some paths
HOME = os.path.expanduser("~")
GEPARD_DIR = os.path.join(os.path.sep, HOME, 'gepard3')
PYPE_DIR = os.path.join(os.path.sep, GEPARD_DIR, 'pype')
NN = os.path.join(HOME, 'neural-net-tf')
CHECKMATE_DIR = os.path.join(os.path.sep, NN, 'checkmate')
sys.path.append(PYPE_DIR)
sys.path.append(CHECKMATE_DIR)

# gepard modules and stuff, including experimental
# data (from abbrevs)
import Model, Approach, Data, utils, plots, Approach_new, Model_new
from results import *
from abbrevs import *

class simpleGK_ImH_ReH_ImHt(Model.GK):
    """
    Class that inherit GK model and overrides its ImE,
    ReE, ReHt, ImEt and ReEt with 0.
    """

    #def ImH(self, pt, xi=0):
        #return 0.

    #def ReH(self, pt, xi=0):
        #return 0.

    def ImE(self, pt, xi=0):
        return 0.

    def ReE(self, pt, xi=0):
        return 0.

```

```
39      #def ImHt(self, pt, xi=0):
40          #return 0.
41
42      def ReHt(self, pt, xi=0):
43          return 0.
44
45      def ImEt(self, pt, xi=0):
46          return 0.
47
48      def ReEt(self, pt, xi=0):
49          return 0.
50
51  mGK = simpleGK_ImH_ReH_ImHt()
52  thGK = Approach.BM10tw2(mGK)
53  thGK.name = 'GK-ImH-ReH-ImHt'
54
55  def mockset(origset, th=thGK, error=0.05, observable='BSDw',
56              harmonic=-1, seed=None):
57          """
58          Return simulated DataSet, using kinematics of origset.
59
60          Args:
61              origset (Dataset): Dataset from which kinematics is
62                  taken.
63              th (Approach): Theory/model that will be used to generate
64                      mean values for observables.
65              error (float): Absolute noise/uncertainty introduced into↘
                    data.
66              observable: Which observable is "measured":
67                      BSS, BSSw, BSD, BSDw, BSA, TSA, BCA, ...
68              harmonic (int): 0 for constant, 1 for cos(phi),
69                      -1 for sin(phi), ...
70              seed (int): Seed if reproducibility wanted.
71          Returns:
72              dataset (Dataset): New simulated dataset.
73          """
74
75      np.random.seed(seed)
76      new = []
77      for origpt in origset:
78          pt = origpt.copy()
79          pt.yaxis = observable
80          pt.y1name = observable
81          pt.units[pt.y1name] = 'nb/GeV^4'  # needed just for plotting
82          pt.FTn = harmonic
83          if observable in ['BSDw', 'BSSw']:
```

```
84              pt.in1polarizationvector = 'L'
85              pt.in1polarization = 1
86              pt.in2polarization = 0
87          if observable == 'TSA':
88              pt.in1polarization = 0
89              pt.in2polarizationvector = 'L'
90              pt.in2polarization = 1
91          if observable == 'BCA':
92              pt.in1polarization = 0
93              pt.in2polarization = 0
94              pt.in1charge = -1
95          # Exact model result will be shifted by gaussian random
96          # error:
97          pt.val = th.predict(pt) + error*np.random.randn()
98          pt.err = error
99          new.append(pt)
100     return Data.DataSet(new)
101
102
103 def get_target_data():
104     return data[101]
105
106
107 def get_dataset(return_fit=False, random_state=4,
108                 datasets=['BSDw', 'BSSw0', 'BSSw1', 'TSA', 'BCA']):
109     """
110     Function that generates mock dataset.
111
112     Args:
113         return_fit (bool): If true, unshuffled
114             mock data is returned.
115         random_state (int): Seed for spliting
116             the dataset.
117          datasets (list): List of names of observables
118             from which to create the dataset.
119
120     Returns:
121         target (DataSet): Dataset of correct GK model
122             predictions containing all used observables.
123         target_BSDw (DataSet): Dataset of correct GK model
124             predictions containing only BSDw observable.
125         target_cffs (dict): Dictionary of GK model CFF
126             predictions for each point in BSDw observable.
127         fitpoints_train (list): List of datapoints used
128             for training.
129         fitpoints_val (list): List of datapoints used
```

```
130             for validation.
131         fit (DataSet): If return_fit is True then a list
132             of unshuffled mock data is returned.
133         random_state (int): Seed for reproducibility.
134 
135     """
136 
137     target_BSDw = mockset(data[101], th=thGK, error=0)
138     fit_BSDw = mockset(data[101], th=thGK, error=0.03,
139                        seed=random_state)
140     target_BSSw0 = mockset(data[101], th=thGK, error=0,
141                            observable='BSSw', harmonic=0)
142     fit_BSSw0 = mockset(data[101], th=thGK, error=0.05,
143                         observable='BSSw', harmonic=0,
144                         seed=random_state+1)
145     target_BSSw1 = mockset(data[101], th=thGK, error=0,
146                            observable='BSSw', harmonic=1)
147     fit_BSSw1 = mockset(data[101], th=thGK, error=0.01,
148                         observable='BSSw', harmonic=1,
149                         seed=random_state+2)
150     target_TSA = mockset(data[101], th=thGK, error=0,
151                          observable='TSA', harmonic=-1)
152     fit_TSA = mockset(data[101], th=thGK, error=0.015,
153                       observable='TSA', harmonic=-1,
154                       seed=random_state+3)
155     target_BCA = mockset(data[101], th=thGK, error=0,
156                          observable='BCA', harmonic=1)
157     fit_BCA = mockset(data[101], th=thGK, error=0.015,
158                       observable='BCA', harmonic=1,
159                       seed=random_state+4)
160 
161     assert len(datasets) > 0
162     mapping_target = {'BSDw': target_BSDw, 'BSSw0': target_BSSw0,
163                       'BSSw1': target_BSSw1, 'TSA': target_TSA,
164                       'BCA': target_BCA}
165     mapping_fit = {'BSDw': fit_BSDw, 'BSSw0': fit_BSSw0,
166                    'BSSw1': fit_BSSw1, 'TSA': fit_TSA,
167                    'BCA': fit_BCA}
168     target = mapping_target[datasets[0]]
169     for name in datasets[1:]:
170         target = target + mapping_target[name]
171 
172     fit = mapping_fit[datasets[0]]
173     for name in datasets[1:]:
174         fit = fit + mapping_fit[name]
175 
```

```
176     target = [pt for pt in target]
177     fit = [pt for pt in fit]
178     fitpoints_train, fitpoints_val = train_test_split(
179         fit, test_size=0.1, random_state=random_state)
180     target_cffs = {'ImH': [thGK.m.ImH(pt) for pt in target_BSDw],
181                    'ReH': [thGK.m.ReH(pt) for pt in target_BSDw],
182                    'ImHt': [thGK.m.ImHt(pt) for pt in target_BSDw]}
183     if return_fit:
184         ret = (target, target_BSDw, target_cffs,
185                fitpoints_train, fitpoints_val, fit)
186     else:
187         ret = (target, target_BSDw, target_cffs,
188                fitpoints_train, fitpoints_val)
189     return ret
```

Dodatak B *models.py*

Ovdje je dan kôd za stvaranje modela CFF funkcija, njihovo povezivanje u funkciju gubitka te stvaranje različitih operacija za treniranje.

Kôd 4: *models.py*

```
1  import os, sys, shelve, logging
2  from shutil import rmtree
3
4  import numpy as np
5  import tensorflow as tf
6  import matplotlib.pyplot as plt
7  from math import sqrt, cos, sin
8  import copy
9
10 # Some paths
11 HOME = os.path.expanduser("~")
12 GEPARD_DIR = os.path.join(os.path.sep, HOME, 'gepard3')
13 PYPE_DIR = os.path.join(os.path.sep, GEPARD_DIR, 'pype')
14 NN = os.path.join(HOME, 'neural-net-tf')
15 CHECKMATE_DIR = os.path.join(os.path.sep, NN, 'checkmate')
16 sys.path.append(PYPE_DIR)
17 sys.path.append(CHECKMATE_DIR)
18
19 # gepard modules and stuff, including experimental data
20 # (from abbrevs)
21 import Model, Approach, Data, utils, plots, Approach_new, Model_new
22 from results import *
23 from abbrevs import *
```

```
24
25
26 LAMBDA = 1.
27
28
29 class ToyModel(Model_new.ComptonFormFactors,
30                Model_new.ElasticDipole):
31     """
32     A model for CFF functions ImH, ReH and ImHt, the rest are zero.
33
34     The code that calculate observables creates new instance of
35     a Point and replaces my placeholders. So CFF functions all
36     have the same input ([point.t, point.xi]) and do not consider
37     pt argument that is passed to them by the Approach.
38     """
39
40     def __init__(self, placeholder,
41                  num_neurons_per_layers=[100, 100],
42                  initializer=tf.keras.initializers.RandomNormal(
43                     stddev=0.1)):
44         """
45         Initialize the model and store placeholders for later usage.
46
47         Args:
48             placeholder (tf.placeholder): A placeholder through which
49                     the CFF functions acquire inputs.
50             num_neurons_per_layers (list): List of numbers of neurons
51                         per each hidden layer.
52             initializer (tf.initializer): Initializer for weights and
53                         biases.
54         """
55         Model_new.ComptonFormFactors.__init__(self)
56         Model_new.ElasticDipole.__init__(self)
57         self.input = placeholder
58         assert len(num_neurons_per_layers) > 0
59         self.num_neurons_per_layers = num_neurons_per_layers
60         self.initializer = initializer
61
62         self._ImH_create()
63         self._ReH_create()
64         self._ImHt_create()
65
66     def _ImH_create(self):
67         """
68         Create neural network for ImH CFF function.
69         """
```

```
70          ImH_model = tf.keras.Sequential(name='ImH')
71
72          for num_neurons in self.num_neurons_per_layers:
73              ImH_model.add(tf.keras.layers.Dense(
74                          num_neurons, activation=tf.nn.tanh,
75                          kernel_initializer=self.initializer,
76                          bias_initializer=self.initializer,
77                          kernel_regularizer=tf.keras.regularizers.l2(
                                LAMBDA)))
78          ImH_model.add(tf.keras.layers.Dense(
79                          1, kernel_initializer=self.initializer,
80                          bias_initializer=self.initializer,
81                          kernel_regularizer=tf.keras.regularizers.l2(
                                LAMBDA)))
82          self._ImH = ImH_model(self.input)
83
84      def ImH(self, pt):
85          """
86          Calculate ImH.
87
88          Args:
89              pt (DataPoint): Unused argument.
90          """
91          return self._ImH
92
93      def _ReH_create(self):
94          """
95          Create neural network for ReH CFF function.
96          """
97          ReH_model = tf.keras.Sequential(name='ReH')
98          for num_neurons in self.num_neurons_per_layers:
99              ReH_model.add(tf.keras.layers.Dense(
100                             num_neurons, activation=tf.nn.tanh,
101                             kernel_initializer=self.initializer,
102                         bias_initializer=self.initializer,
103                             kernel_regularizer=tf.keras.
                                    regularizers.l2(LAMBDA)))
104         ReH_model.add(tf.keras.layers.Dense(
105                         1, kernel_initializer=self.initializer,
106                         bias_initializer=self.initializer,
107                         kernel_regularizer=tf.keras.regularizers.l2(
                                LAMBDA)))
108         self._ReH = ReH_model(self.input)
109
110     def ReH(self, pt):
111         """
```

```
112         Calculate ReH.
113
114         Args:
115             pt (DataPoint): Unused argument.
116         """
117         return self._ReH
118
119     def _ImHt_create(self):
120         """
121         Create neural network for ImHt CFF function.
122         """
123         ImHt_model = tf.keras.Sequential(name='ImHt')
124         for num_neurons in self.num_neurons_per_layers:
125             ImHt_model.add(tf.keras.layers.Dense(
126                         num_neurons, activation=tf.nn.tanh,
127                                 kernel_initializer=self.initializer,
128                                 bias_initializer=self.initializer,
129                                 kernel_regularizer=tf.keras.↘
                                        regularizers.l2(LAMBDA)))
130         ImHt_model.add(tf.keras.layers.Dense(
131                         1, kernel_initializer=self.initializer,
132                         bias_initializer=self.initializer,
133                         kernel_regularizer=tf.keras.regularizers.l2(↘
                            LAMBDA)))
134         self._ImHt = ImHt_model(self.input)
135
136     def ImHt(self, pt):
137         """
138         Calculate ImHt.
139
140         Args:
141             pt (DataPoint): Unused argument.
142         """
143         return self._ImHt
144
145 def create_graph(
146                 point_example, lr=1., reg_constant=0.0,
147                 num_neurons_per_layers=[100, 100],
148                 initializer=tf.keras.initializers.RandomNormal(stddev↘
                    =0.1)):
149     """
150     Create graph for learning the CFF functions.
151
152     Args:
153         point_example (DataPoint): One datapoint from some
154                         dataset for the placeholder to copy.
```

```
155         lr (float): Learning rate.
156         reg_constant (float): Regularization constant.
157         num_neurons_per_layers (list): List of numbers of
158                         neurons per each hidden layer.
159         initializer (tf.initializer): Initializer for weights and ↘
               biases.
160 
161     Returns:
162         point (tf.placeholder): Point placeholder.
163         y_true (tf.placeholder): placeholder for the true value of
164             observable for a given point.
165         partial_loss (tf.Tensor): Partial loss function tensor.
166         train_ops (dict): A dictionary containing training operations
167                         for different combinations of CFF functions.
168         predictions (dict): Dictionary of predictions for each CFF
169                         function.
170         global_step_tensor (tf.Tensor): Global step tensor.
171     """
172     tf.reset_default_graph()
173     with tf.device('/cpu:0'):
174 
175         point = copy.copy(point_example) # can be any point, I think
176         # placeholders for inputs
177         point.t = tf.placeholder(tf.float32, name='t')
178         point.xB = tf.placeholder(tf.float32, name='xB')
179         point.y1name = tf.placeholder(tf.string, name='y1name')
180         point.FTn = tf.placeholder(tf.float32, name='FTn')
181         point.in1polarization = tf.placeholder(
182                         tf.float32, name='in1polarization')
183         point.in2polarization = tf.placeholder(
184                         tf.float32, name='in2polarization')
185         point.in2polarizationvector = tf.placeholder(
186                         tf.string, name='in2pvector')
187         point.W = tf.placeholder(tf.float32, name='W')
188         point.Q2 = tf.placeholder(tf.float32, name='Q2')
189         point.eps = tf.placeholder(tf.float32, name='eps')
190         point.yaxis = tf.placeholder(tf.string, name='yaxis')
191 
192         point.tm = -point.t
193         point.xi = point.xB / (2. - point.xB)
194 
195         # Create a model given the input data.
196         # The same point goes to the model to calculate the CFF
197         # functions and to the th.predict(point). This couldn't be
198         # solved differently because the code that calculate
199         # observables creates new instance of a Point and
```

```
200         # replaces my placeholders.  So CFF functions all have
201         # the same input ([point.t, point.xi])  and do not consider
202         # pt argument that is passed to them by the Approach.
203         m = ToyModel(tf.reshape(tf.stack([point.t,
204                                           point.xi],0),(1,2)))
205         th = Approach_new.BM10tw2(m)
206         th.prepare(point)
207
208         y_true = tf.placeholder(tf.float32, name='y_true')
209         y_hat = th.predict(point)
210
211         predictedImH = th.m.ImH(point)
212         predictedReH = th.m.ReH(point)
213         predictedImHt = th.m.ImHt(point)
214
215         reg_losses = tf.get_collection(
216                         tf.GraphKeys.REGULARIZATION_LOSSES)
217
218         # Definition of loss function
219         partial_loss = tf.square(y_hat - y_true,
220                                  name='partial_loss') + \
221                            reg_constant*sum(reg_losses)
222
223         global_step_tensor = tf.train.get_or_create_global_step()
224
225         # Enable training of subsets of CFF functions
226         cffs = ['ImH', 'ReH', 'ImHt']
227         var_dic = {cff:tf.trainable_variables(cff+"/") for cff in
                ↘ cffs}
228
229         var_list = list()
230         lens = list()
231         for key in cffs:
232             var_list = var_list + var_dic[key]
233             lens.append(len(var_dic[key]))
234
235         all_grads = tf.gradients(partial_loss, var_list)
236
237         grads = dict()
238         index = 0
239         for cff, size in zip(cffs, lens):
240             grads[cff] = all_grads[index:index+size]
241             index += size
242
243         train_op1 = tf.train.AdamOptimizer().apply_gradients(
244             zip(grads['ImH']+grads['ImHt'],
```

```
                var_dic['ImH']+var_dic['ImHt']),
            global_step=global_step_tensor)
        train_op2 = tf.train.AdamOptimizer().apply_gradients(
            zip(grads['ImH']+grads['ReH'],
                var_dic['ImH']+var_dic['ReH']),
            global_step=global_step_tensor)
        train_op3 = tf.train.AdamOptimizer().apply_gradients(
            zip(grads['ImHt']+grads['ReH'],
                var_dic['ImHt']+var_dic['ReH']),
            global_step=global_step_tensor)

        train_op_all = tf.train.AdamOptimizer().minimize(
            partial_loss, global_step=global_step_tensor)
        predictions = {'ImH': predictedImH,
                       'ReH': predictedReH,
                       'ImHt': predictedImHt}
        train_ops = {'ImH_ImHt': train_op1,
                     'ImH_ReH': train_op2,
                     'ReH_ImHt': train_op3,
                     'ImH_ImHt_ReH': train_op_all,}
        return point, y_true, partial_loss, train_ops, \
                          predictions, global_step_tensor
```

Dodatak C train.py

Ovdje je dan prikaz kôda za treniranje i validaciju modela.

Kôd 5: *train.py*

```
import os, sys, shelve, logging
from shutil import rmtree

import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from math import sqrt, cos, sin
import copy
import checkmate

# Some paths
HOME = os.path.expanduser("~")
GEPARD_DIR = os.path.join(os.path.sep, HOME, 'gepard3')
PYPE_DIR = os.path.join(os.path.sep, GEPARD_DIR, 'pype')
NN = os.path.join(HOME, 'neural-net-tf')
CHECKMATE_DIR = os.path.join(os.path.sep, NN, 'checkmate')
```

```
17 sys.path.append(PYPE_DIR)
18 sys.path.append(CHECKMATE_DIR)
19
20 # gepard modules and stuff, including
21 # experimental data (from abbrevs)
22 import Model, Approach, Data, utils, plots, Approach_new, Model_new
23 from results import *
24 from abbrevs import *
25
26
27 def create_feed(placeholder, y_true, datapoint):
28     """
29     Create feed from datapoint to placeholders.
30
31     Args:
32         placeholder (tf.placeholder): Datapoint
33             placeholder (without val attribute).
34         y_true (tf.placeholder): placeholder for val
35             attribute of the datapoint.
36         datapoint (DataPoint): Datapoint to
37             feed the network.
38
39     Returns:
40         feed (dict): Feed dictionary.
41     """
42     feed = {placeholder.in2polarization:
43                 float(datapoint.in2polarization),
44             placeholder.t: datapoint.t,
45             placeholder.xB: datapoint.xB,
46             placeholder.y1name: datapoint.y1name,
47             placeholder.FTn: float(datapoint.FTn),
48             placeholder.in1polarization:
49                 float(datapoint.in1polarization),
50             placeholder.in2polarizationvector:
51                 datapoint.in2polarizationvector
52                     if hasattr(datapoint,
53                                'in2polarizationvector')
54                     else '',
55             placeholder.W: datapoint.W,
56             placeholder.Q2: datapoint.Q2,
57             placeholder.eps: datapoint.eps,
58             placeholder.yaxis: datapoint.yaxis,
59             y_true: datapoint.val}
60     return feed
61
62
```

```
63  def train(placeholder, y_true, global_step_tensor,
64            partial_loss, train_op, fitpoints_train,
65            fitpoints_val, save_dir, num_epochs=50,
66            fine_tune_from=None, best_checkpoints_to_keep=1,
67            logdir='logs'):
68      """
69      Train the model.
70
71      Args:
72          placeholder (tf.placeholder): Datapoint placeholder
73              (without val attribute).
74          y_true (tf.placeholder): placeholder for val attribute
75              of the datapoint.
76          global_step_tensor (tf.Tensor): Global step tensor.
77          partial_loss (tf.Tensor): Loss function tensor.
78          train_op (tf.Op): Training operation to execute each
79              step.
80          fitpoints_train (list): List of training points.
81          fitpoints_val (list): List of points for valudation.
82          save_dir (str): Path to the directory where the
83              checkpoints will be saved.
84          num_epochs (int): Number of epochs to train.
85          fine_tune_from (str): Path to the directory of the
86              checkpoint from which to fine tune.
87          best_checkpoints_to_keep (int): Number of best
88              checkpoints on validation set to keep.
89          logdir (str): Path to the directory to where to
90              save the logs and tensorboard data.
91      Returns:
92          losses (list): List of losses on training set.
93          val_losses (list): List of losses on validation set.
94      """
95      if os.path.exists(save_dir):
96          if len(os.listdir(save_dir)) != 0:
97              raise AssertionError('save_dir should be empty!')
98
99      saver = checkmate.BestCheckpointSaver(
100       save_dir=save_dir,
101       num_to_keep=best_checkpoints_to_keep,
102       maximize=False
103     )
104     loader = tf.train.Saver()
105
106     losses, val_losses = list(), list()
107     with tf.Session() as sess:
108         sess.run(tf.global_variables_initializer())
```

```
109         if fine_tune_from is not None:
110             loader.restore(sess, checkmate.get_best_checkpoint(
111                 fine_tune_from, select_maximum_value=False,
112                 index=0))
113         writer = tf.summary.FileWriter(logdir, sess.graph)
114 
115         for epoch in range(num_epochs):
116             # Training
117             for pt in fitpoints_train :
118                 feed = create_feed(placeholder, y_true, pt)
119                 _ = sess.run([train_op], feed_dict=feed)
120 
121             # Calculate loss on train and validation set
122             loss = 0
123             for pt in fitpoints_train :
124                 feed = create_feed(placeholder, y_true, pt)
125                 pl = sess.run(partial_loss, feed_dict=feed)
126                 loss += pl
127 
128             loss_val = 0
129             for pt in fitpoints_val :
130                 feed = create_feed(placeholder, y_true, pt)
131                 pl = sess.run(partial_loss, feed_dict=feed)
132                 loss_val += pl
133 
134             print ("{}. epoch: loss = {}".format(epoch, loss))
135             losses.append(loss)
136             print ("\t Loss on validation set: {}.".format(loss_val))
137             val_losses.append(loss_val)
138             saver.handle(loss_val, sess,  global_step_tensor)
139         writer.close()
140         return losses, val_losses
```

Dodatak D Primjer *Jupyter* bilježnice za učenje neuronske mreže

example.ipynb

[1]:
```
import os, sys
HOME = os.path.expanduser("~")
NN = os.path.join(HOME, 'neural-net-tf')
CHECKMATE_DIR = os.path.join(os.path.sep, NN, 'checkmate')
sys.path.append(NN)
sys.path.append(CHECKMATE_DIR)

from metrics import rmse
import mock_data
import models
import train
import visualization
import matplotlib.pyplot as plt

%env CUDA_DEVICE_ORDER=PCI_BUS_ID
%env CUDA_VISIBLE_DEVICES=0

FIG_NUM = 0
```

```
env: CUDA_DEVICE_ORDER=PCI_BUS_ID
env: CUDA_VISIBLE_DEVICES=0
```

[2]:
```
target, minimal_target, target_cffs, fitpoints_train, fitpoints_val = \
mock_data.get_dataset(random_state=1, datasets=['BSDw', 'BSSw0', 'BSSw1',␣
 ↪'TSA', 'BCA'])
```

[3]:
```
placeholder, y_true, partial_loss, train_ops, \
predictions, global_step_tensor = models.create_graph(fitpoints_train[0],
                                                      num_neurons_per_layers =␣
 ↪[50]*100,
                                                      reg_constant=0.05)
```

[4]:
```
save_dir = '/home/icoric/num_data/40'
```

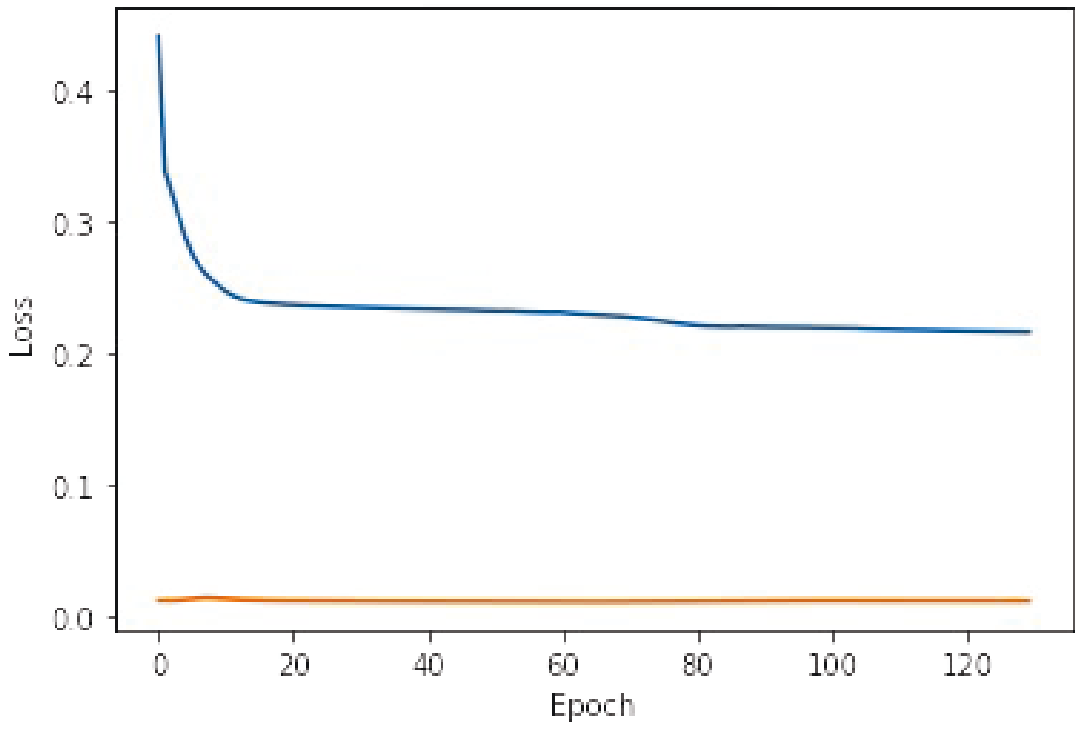
[5]:
```
num_epochs = 130
losses, val_losses = train.train(placeholder, y_true, global_step_tensor,
                                 partial_loss,
                                 train_ops['ImH_ImHt_ReH'],
```

```
                                fitpoints_train, fitpoints_val,
                                save_dir,
                                num_epochs=num_epochs)

plt.figure(FIG_NUM)
FIG_NUM += 1

plt.title("Loss on training set")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.plot(range(num_epochs), losses)
plt.plot(range(num_epochs), val_losses)
```

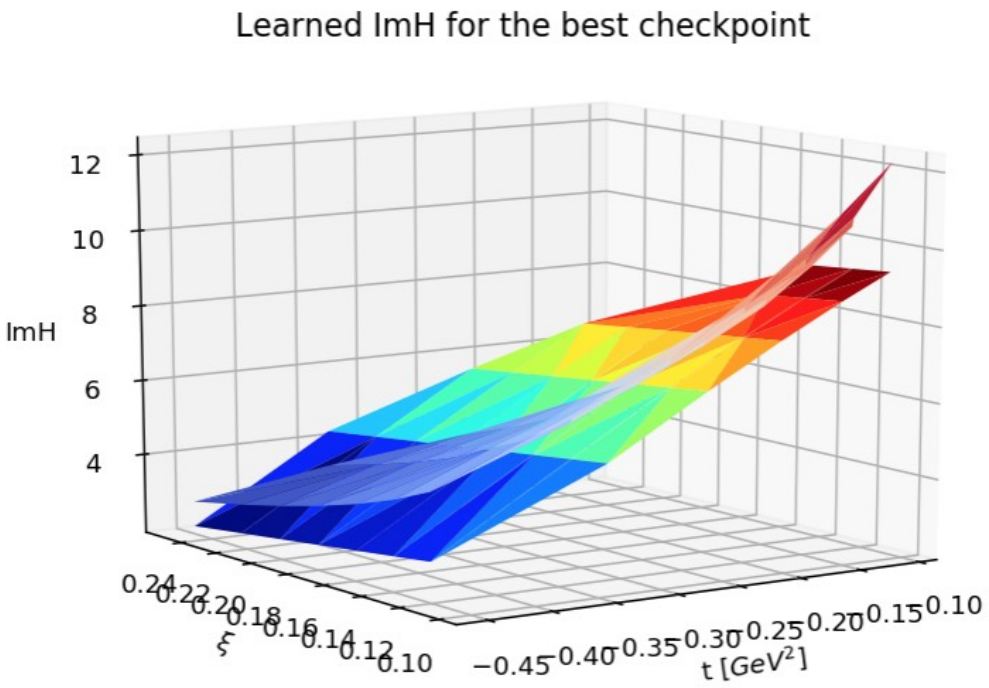
```
0. epoch: loss = [0.44162652]
         Loss on validation set: [0.01180452].
INFO:tensorflow:best.ckpt-216 is not in all_model_checkpoint_paths. Manually
adding it.
1. epoch: loss = [0.33880597]
         Loss on validation set: [0.01171755].
INFO:tensorflow:best.ckpt-432 is not in all_model_checkpoint_paths. Manually
adding it.
2. epoch: loss = [0.32134378]
         Loss on validation set: [0.01176768].
3. epoch: loss = [0.30446875]
         Loss on validation set: [0.01188274].
4. epoch: loss = [0.28851244]
         Loss on validation set: [0.01230035].
5. epoch: loss = [0.275816]
         Loss on validation set: [0.01295693].
6. epoch: loss = [0.26689762]
         Loss on validation set: [0.013552].
7. epoch: loss = [0.260569]
         Loss on validation set: [0.01384054].
8. epoch: loss = [0.25562844]
         Loss on validation set: [0.01375869].
9. epoch: loss = [0.25119]
         Loss on validation set: [0.01339135].
10. epoch: loss = [0.24720092]
         Loss on validation set: [0.01292786].
11. epoch: loss = [0.24407019]
         Loss on validation set: [0.01252606].
12. epoch: loss = [0.24189906]
         Loss on validation set: [0.01223819].
13. epoch: loss = [0.2404665]
         Loss on validation set: [0.01204983].
.
.
.
```

```
.
.
.

121. epoch: loss = [0.21681468]
        Loss on validation set: [0.01148321].
122. epoch: loss = [0.21678075]
        Loss on validation set: [0.01148294].
123. epoch: loss = [0.2167309]
        Loss on validation set: [0.01148486].
124. epoch: loss = [0.21669625]
        Loss on validation set: [0.01148621].
125. epoch: loss = [0.21656965]
        Loss on validation set: [0.01147804].
126. epoch: loss = [0.21648073]
        Loss on validation set: [0.01147649].
127. epoch: loss = [0.21656023]
        Loss on validation set: [0.01149054].
128. epoch: loss = [0.21650629]
        Loss on validation set: [0.01149148].
129. epoch: loss = [0.21627666]
        Loss on validation set: [0.01147378].
```

[5]: `[<matplotlib.lines.Line2D at 0x7fc46bf02048>]`

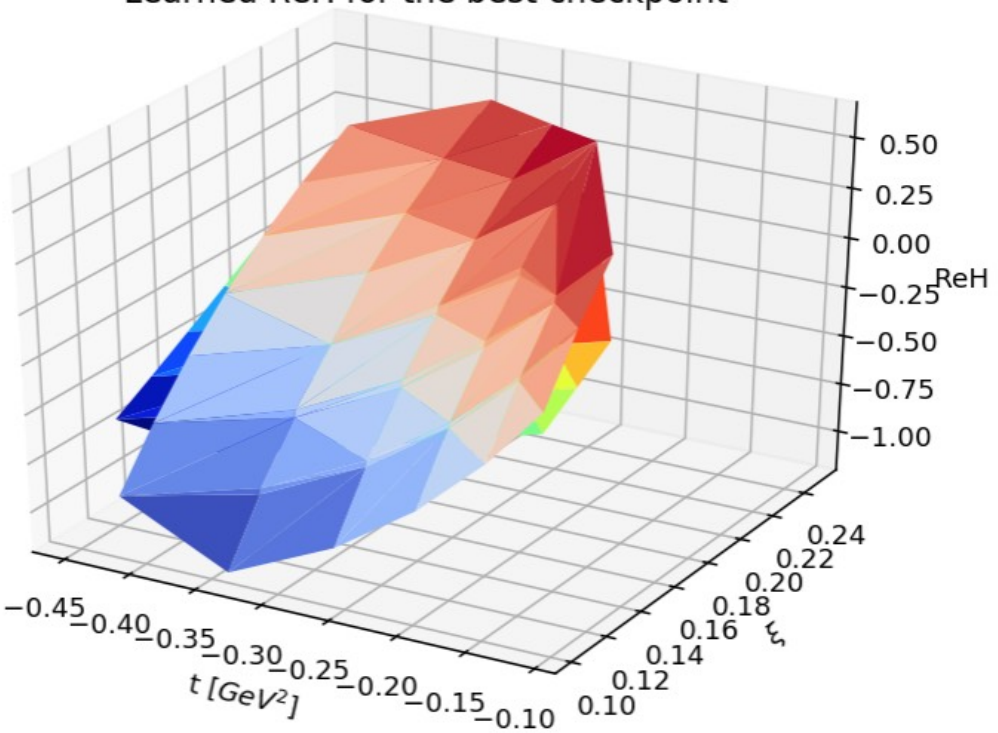


[6]:
```
%matplotlib notebook

fig = plt.figure(FIG_NUM)
FIG_NUM += 1
cff = 'ImH'
print(cff, rmse(placeholder, predictions[cff], target_cffs[cff],␣
 ↪minimal_target, save_dir))
visualization.visualize_predictions(fig, minimal_target, placeholder,
                                    predictions[cff], cff, save_dir,
                                    target_cffs[cff])
```



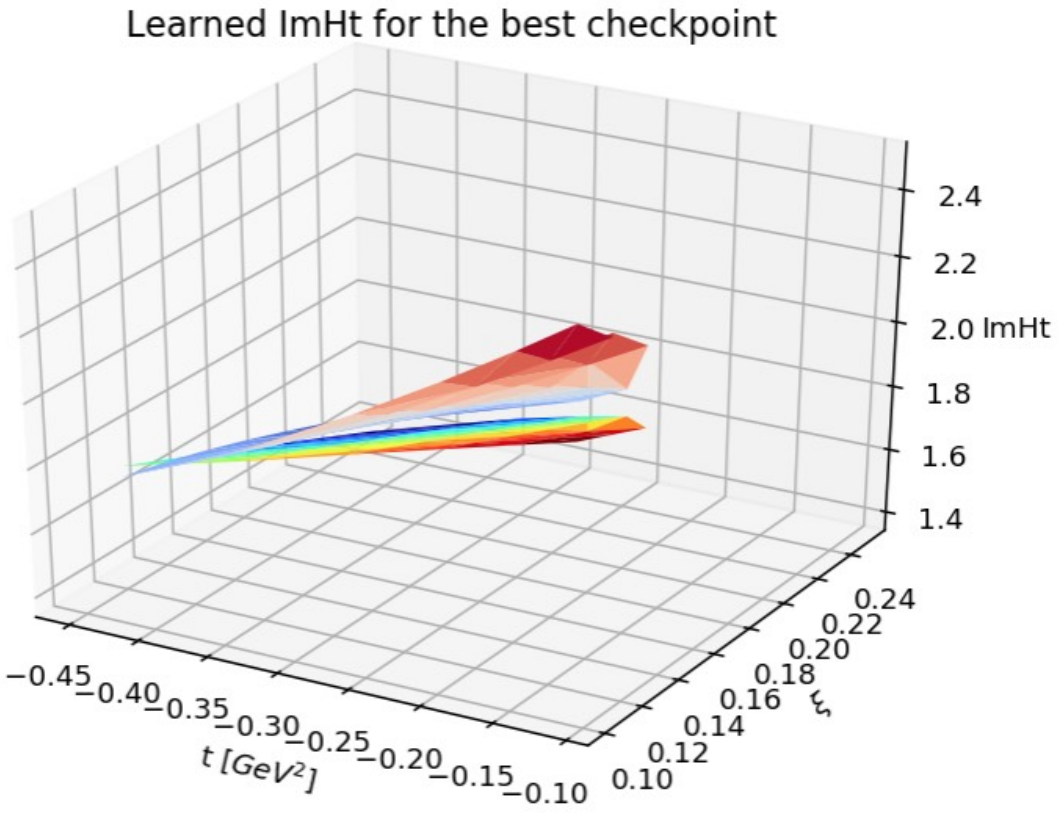

```
INFO:tensorflow:Restoring parameters from
/home/icoric/num_data/40/best.ckpt-14040
ImH 1.1240124913814344
INFO:tensorflow:Restoring parameters from
/home/icoric/num_data/40/best.ckpt-14040
```

[7]:
```
fig = plt.figure(FIG_NUM)
FIG_NUM += 1
cff = 'ReH'
print(cff, rmse(placeholder, predictions[cff], target_cffs[cff],␣
 ↪minimal_target, save_dir))
visualization.visualize_predictions(fig, minimal_target, placeholder,
                                    predictions[cff], cff, save_dir,
                                    target_cffs[cff])
```




```
INFO:tensorflow:Restoring parameters from
/home/icoric/num_data/40/best.ckpt-14040
ReH 0.23561254850798585
INFO:tensorflow:Restoring parameters from
/home/icoric/num_data/40/best.ckpt-14040
```

[8]:
```
fig = plt.figure(FIG_NUM)
FIG_NUM += 1
cff = 'ImHt'
print(cff, rmse(placeholder, predictions[cff], target_cffs[cff],␣
 ↪minimal_target, save_dir))
visualization.visualize_predictions(fig, minimal_target, placeholder,
                                    predictions[cff], cff, save_dir,
                                    target_cffs[cff])
```




```
INFO:tensorflow:Restoring parameters from
/home/icoric/num_data/40/best.ckpt-14040
ImHt 0.14249140712213215
INFO:tensorflow:Restoring parameters from
/home/icoric/num_data/40/best.ckpt-14040
```

Literatura

[1] Müller D., Robaschik D., Geyer B., Dittes F. M. i Hořejši J., Fortschr. Phys. **42**, 101 (1994), hep-ph/9812448

[2] Radyushkin A. V., Phys. Lett. **380**, 417 (1996), hep-ph/9604317

[3] Ji X.-D., Phys. Rev. **D55**, 7114 (1997), hep-ph/9609381

[4] Diehl M., Generalized Parton Distributions, Phys. Rept. 388 (2003) 41-277, hep-ph/0307382

[5] Kumerički K.; Liuti S.; Moutarde H. *GPD phenomenology and DVCS fitting.* Eur. Phys. J. A, 52 (2016)

[6] Atoms, Molecules, Elements, Compounds https://brilliant.org/wiki/atoms-molecules-elements-compounds, [7. 7. 2019.]

[7] *Artifical Neural Networks and Neural Networks Applications* https://www.xenonstack.com/blog/artificial-neural-network-applications, [19. 6. 2019.]

[8] *Introduction to Neural Networks* https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-c50f6012d5eb, [19. 6. 2019.]

[9] *Activation functions.* https://www.julyedu.com/ question/big/kp id/26/ques id/1044. [19.6. 2019.]

[10] *Artifical Neural Networks - Quora* https://www.quora.com/How-does-deep-learning-work-and-how-is-it-different-from-normal-neural-networks-applied-with-SVM-How-does-one-go-about-starting-to-understand-them-papers-blogs-articles, [7. 7. 2019.]

[11] *Batch gradient descent vs. Stochastic gradient descent* https://wikidocs.net/3413, [20. 6. 2019.]

[12] Csáji B. C. *Approximation with Artificial Neural Networks*; Faculty of Sciences; Eötvös Loránd University, Hungary (2001)

[13] *What is underfitting and overfitting in machine learning and how to deal with it.* https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76, [7. 7. 2019.]

[14] *Machine Learning Explained* http://mlexplained.com/2018/04/24/overfitting-isnt-simple-overfitting-re-explained-with-priors-biases-and-no-free-lunch, [20. 6. 2019.]

[15] García S. I. L0 Norm, L1 Norm, L2 Norm & L-Infinity Norm https://medium.com/@montjoile/l0-norm-l1-norm-l2-norm-l-infinity-norm-7a7d18a4f40c, [7. 7. 2019.]

[16] Kroll P., Moutarde H., Sabatié F. *From hard exclusive meson electroproduction todeeply virtual Compton scattering* Eur. Phys. J. C73 (2013) no.1, 2278, arXiv:1210.6975 [hep-ph]

[17] CLAS detector https://en.wikipedia.org/wiki/CLAS_detector, [7. 7. 2019.]

[18] Goloskokov S. G. i Kroll P., Eur. Phys. J. C **65**, 137 (2010) arXiv:1106.4897 [hep-ph]

[19] Kingma D. P., Ba J. Adam: A Method for Stochastic Optimization arXiv:1412.6980 [cs.LG]