Proučavanje duboko virtualnog komptonskog raspršenja pomoću strojnog učenja

Cvitković, Marko

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:651085

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-24**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

Marko Cvitković

PROUČAVANJE DUBOKO VIRTUALNOG KOMPTONSKOG RASPRŠENJA POMOĆU STROJNOG UČENJA

Diplomski rad

Zagreb, 2020.

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

INTEGRIRANI PREDDIPLOMSKI I DIPLOMSKI SVEUČILIŠNI STUDIJ
FIZIKA; SMJER ISTRAŽIVAČKI

Marko Cvitković

Diplomski rad

Proučavanje duboko virtualnog komptonskog raspršenja pomoću strojnog učenja

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

Ocjena diplomskog rada: _____

Povjerenstvo: 1. _____

2. _____

3. _____

Datum polaganja: _____
Zagreb, 2020.

Zahvaljujem mentoru prof. dr. sc. Krešimiru Kumeričkom na savjetima i velikoj pomoći pri izradi ovog rada.
Zahvaljujem Josipi, obitelji i prijateljima na svesrdnoj podršci tijekom studija.

Sažetak

Duboko virtualno komptonsko raspršenje (DVCS) je proces kojim se dobro može izučavati trodimenzionalna raspodjela kvarkova i gluona u protonu. Funkcije strukture tog procesa (komptonski form faktori, CFF-ovi) su esencijalno neperturbativni objekti koje modeliramo neuronskim mrežama. One predstavljaju optimalan izbor zbog činjenice da ne unose pristranost. U ovom radu, tako generiran model DVCS amplituda prilagođava se različitim eksperimentalnim mjerenjima. Pokazano je da je prilagodba na podatke najkvalitetnija u slučaju kada se koriste svi CFF-ovi u vodećem doprinosu. Uočene su opservable na koje se model kvalitetnije prilagodi u odnosu na druge te su prikazani naučeni CFF-ovi ovisno o izboru značajki neuronske mreže.

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

Study of deeply virtual Compton scattering using machine learning

Abstract

Deeply virtual Compton scattering (DVCS) is a process which provides access to quality research of three-dimensional distribution of quarks and gluons inside the proton. Structure functions of that process (Compton form factors, CFF) are essentially non-perturbative objects which we model with neural networks. They represent the optimal choice due to the fact that they do not introduce bias. In this thesis, DVCS amplitudes generated that way are fitted to different sets of experimental data. It is shown that using all leading order CFF functions in modeling experimental observables yields best fit to the data. Also, observables which yield better fits to data than the others are notices and learned CFF functions, based on neural network features choices, are shown.

Keywords: Deeply virtual Compton scattering, Generalized parton distributions, Compton form factors, Neural networks.

Sadržaj

1 Uvod **1**

2 Teorijski pregled **3**

2.1 Nukleonski form faktori 3
2.2 Partonske distribucijske funkcije 4
2.3 Generalizirane partonske distribucije 7
2.4 Komptonski form faktori 11
2.5 DVCS opservable 13

3 Umjetne neuronske mreže **16**

3.1 Motivacija i povijesni uvod 16
3.2 Umjetni neuron - perceptron 17
3.3 Neuronske mreže 18
3.4 Aproksimacijska svojstva neuronskih mreža 20
3.5 Odabir modela 21
3.6 Optimizacija parametara i gradijentni spust 23
3.7 Algoritam propagacije unatrag 25
3.8 Problemi pri učenju dubokih mreža 27

4 Ekstrakcija CFF funkcija **29**

4.1 Umjetni podaci 29
4.2 Izbor arhitekture i značajki mreže 30
4.3 Rezultati prilagodbe neuronskih mreža na opservable 36
4.4 Ekstrahirane CFF funkcije 40

5 Zaključak **47**

Dodaci **48**

A load_data.py **48**

B models.py **50**

C train.py **62**

D visualize_CFFs.py **64**

Literatura **69**

1 Uvod

Struktura materije predmet je ljudskog istraživanja već više od dvije tisuće godine te unatoč tome što je napravljen značajan napredak od koncepta atoma do danas, puno toga nam je još uvijek ostalo nepoznato. Danas standardni model elementarnih čestica dijelimo na temeljne čestice spina 1/2 ili fermione, prijenosnike sila spina 1 ili baždarne bozone te jedno skalarno polje spina 0 ili Higgsov bozon.

Fermione, odnosno 12 čestica za koje se smatra da su konstituenti sve materije u svemiru dalje dijelimo na kvarkove i leptone. Izuzevši 3 tipa električno neutralnih neutrina, sve čestice su električki nabijene i sudjeluju u elektromagnetskim interakcijama izmjenama fotona koje su opisane kvantnom elektrodinamikom (eng. *Quantum Electrodynamics*, QED).

S druge strane, samo kvarkovi nose ekvivalent električnog naboja za jaku interakciju, novi kvantni broj koji nazivamo *boja*. Kvarkove u prirodi ne nalazimo kao slobodne čestice, već uvijek u vezanim stanjima zajedno sa pripadnim baždarnim bozonima (gluonima), kao što su proton ili neutron, koja nazivamo *hadroni* i čije interakcije opisuje kvantna kromodinamika (engl. *Quantum Chromodynamics*, QCD). Zbog ovisnosti jakosti jake sile o energetskoj skali interakcije, nije moguće koristiti perturbativni račun kod QCD pri niskim energijama, stoga se moramo okrenuti eksperimentima za potpunu sliku hadrona i njihove strukture.

Veliki napori već su uloženi u proučavanje eksperimenata koristeći elektrone i poznatu QED interakciju: elastično raspršenje elektrona na protonu i duboko neelastično raspršenje (engl. *Deep Inelastic Scattering*, DIS) iz kojih je moguće izvući nukleonske *form faktore* i *partonske distribucijske funkcije* (engl. *Parton Distribution Functions*, PDFs) koje i dalje ne daju potpunu kvalitativnu informaciju o strukturi hadrona.

Zbog toga su 90-ih godina prošloga stoljeća uvedene *generalizirane partonske distribucije* (engl. *Generalized Parton Distributions*, GPDs) koje daju obje navedene informacije i njihove korelacije te tako efektivno predstavljaju 3D raspodjelu kvarkova i gluona unutar hadrona. Općenito, GPD-ovi se mogu proučavati kroz duboko ekskluzivne procese elektroprodukcije od kojih je najjednostavniji *duboko virtualno komptonsko raspršenje* (engl. *Deeply Virtual Compton Scattering*, DVCS), a kroz koji su direktno dostupni *komptonski form faktori* (engl. *Compton Form Factors*, CFFs), koji

su konvolucije GPD-ova i poznatih perturbativnih amplituda i predstavljaju važan međukorak u cilju poznavanja hadronske strukture.

Koristeći činjenicu da ne unose pristranost, neuronske mreže su se pokazale kao dobar izbor za modeliranje CFF-ova. U ovom radu prikazat će se naučenih 8 CFF funkcija u vodećem doprinosu ovisno o postavkama neuronske mreže i za različite setove mjerenja udarnih presjeka.

U drugom poglavlju dan je teorijski pregled i svojstva GPD-ova, CFF-ova te samog DVCS procesa. Opisane su i različite mjerene opservable koje služe kao podaci za treniranje neuronskih mreža. U trećem poglavlju opisane su neuronske mreže, motivacija njihovog nastanka, sposobnost aproksimacije proizvoljne funkcije i algoritam koji im omogućava učenje - propagacija unatrag (engl. *backpropagation*). Objašnjene su i teškoće koje se mogu pojaviti pri treniranju duboke mreže kao što su *iščezavajući gradijenti* (engl. *vanishing gradients*). U četvrtom poglavlju prikazani su rezultati kvalitete prilagodbe naučenih modela na korištene podatke standardnim χ^2 testom, kao i rezultirajuće ekstrahirane CFF funkcije za sve promatrane neuralne arhitekture i setove podataka.

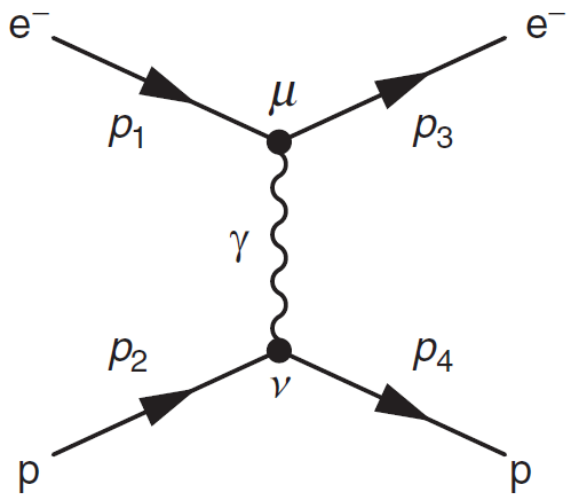
2 Teorijski pregled

2.1 Nukleonski form faktori

Elektron-proton $e^-p \to e^-p$ proces se na niskim energijama opisuje elastičnim raspršenjem (slika 2.1), što znači da su čestice prisutne u početnom i konačnom stanju jednake. Tada je aproksimacija protona kao točkaste čestice valjana, a uz pretpostavke da je odboj protona zanemariv i da je elektron relativističan, diferencijalni udarni presjek dan je sa tzv. Mottovim udarnim presjekom:

$$\left(\frac{d\sigma}{d\Omega}\right)_{Mott} = \frac{\alpha^2}{4E^2\sin^4(\theta/2)}\cos^2\frac{\theta}{2} \tag{2.1}$$

gdje je θ kut raspršenja elektrona, E je energija snopa elektrona, dok je α konstanta fine strukture.



Slika 2.1: Feynmanov dijagram $e^-p \to e^-p$ elastičnog raspršenja. Posuđeno iz [1].

Na višoj energetskoj skali, valna duljina izmijenjenog fotona je manja te proces više nije isključivo elektrostatičke prirode. Proton se više ne smatra točkastim pa se javlja potreba za uvođenjem form faktora koje kvalitativno možemo opisati kao funkcije koje korigiraju razlike u fazi između doprinosa raspršenom valu od strane različitih točaka distribucije naboja. Može se pokazati da je najopćenitiji Lorenz-invarijantni izraz za elektron-proton elastično raspršenje uz izmjenu jednog fotona dan s Rosenbluthovom formulom:

$$\frac{d\sigma}{d\Omega} = \frac{\alpha^2}{4E_1^2\sin^4(\theta/2)}\frac{E_3}{E_1}\left(\frac{G_E^2+\tau G_M^2}{1+\tau}\cos^2\frac{\theta}{2} + 2\tau G_M^2\sin^2\frac{\theta}{2}\right), \tag{2.2}$$

gdje je $\tau = Q^2/4m_p^2$, a Q^2 nazivamo *virtualnost* i interpretiramo kao skala na kojoj ispitujemo strukturu nukleona: veće vrijednosti omogućavaju istraživanje manjih udaljenosti i struktura. $G_E(Q^2)$ i $G_M(Q^2)$ nazivamo Sachsovim električnim i magnetskim form faktorima, funkcije su kvadriranog četveroimpulsa virtualnog fotona i ne mogu se direktno interpretirati kao „klasični" form faktori tipa $F(\mathbf{q}^2)$ koji su funkcije vektora troimpulsa.

Međutim, u sustavu beskonačnog impulsa (Breitovom sustavu) kod elastičnog raspršenja gdje ulazni elektron ima impuls $\vec{k} = +\vec{q}/2$ i proton $\vec{p} = -\vec{q}/2$, navedeni Sachsovi form faktori mogu se interpretirati kao Fourierovi transformati distribucija naboja i magnetskog momenta protona u transverzalnoj ravnini

$$\begin{aligned} G_E(Q^2) &\approx G_E(\vec{q}^2) = \int e^{i\vec{q}\cdot\vec{r}}\rho(\vec{r})d^3\vec{r}, \\ G_M(Q^2) &\approx G_M(\vec{q}^2) = \int e^{i\vec{q}\cdot\vec{r}}\mu(\vec{r})d^3\vec{r}. \end{aligned} \tag{2.3}$$

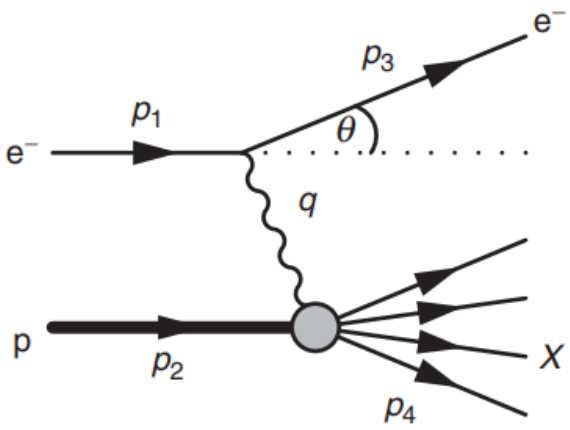
Zbog konzistentnosti s eksperimentalno izmjerenim *anomalnim* magnetskim momentom protona, distribucija magnetskog momenta normalizira se na vrijednost $+2.79$ te stoga vrijedi

$$\begin{aligned} G_E(0) &= \int \rho(\vec{r})d^3\vec{r} = 1, \\ G_M(0) &= \int \mu(\vec{r})d^3\vec{r} = +2.79. \end{aligned} \tag{2.4}$$

Iz poznatih Sachsovih form faktora mogu se dobiti izrazi za nukleonske radijuse naboja i magnetske radijuse. Popularna zagonetka radijusa naboja protona je razlog zbog kojeg je ovo područje i dalje u velikom fokusu istraživanja. Pokušavaju se objasniti neusklađenosti iz različitih metoda mjerenja. Jedna od mogućnosti je pogrešna ekstrapolacija za $G_E(Q^2)$ pri $Q^2 = 0$ zbog čega se elastični form faktori i dalje mjere pri ekstremno niskim vrijednostima Q^2 s velikom preciznošću.

2.2 Partonske distribucijske funkcije

Zbog konačne veličine protona, $G_E(Q^2)$ i $G_M(Q^2)$ postaju maleni za velike vrijednosti Q^2 i elastični diferencijalni udarni presjek rapidno pada. Posljedično, dominantan proces kod visokoenergetskih elektron-proton raspršenja je neelastično raspršenje (slika 2.2) gdje virtualni foton međudjeluje s kvarkovima unutar protona umjesto s protonom kao cjelinom.



Slika 2.2: Feynmanov dijagram $e^-p \to e^-p$ neelastičnog raspršenja. Posuđeno iz [1].

Konačno hadronsko stanje sastoji se od mnogo čestica. Invarijantna masa W tog sustava ovisi o četveroimpulsu virtualnog fotona q za razliku od elastičnog raspršenja gdje je jednaka masi protona. Zbog te činjenice, vidimo da su za opis kinematike neelastičnog raspršenja potrebne dvije fizikalne veličine kako bi se uračunao taj dodatni stupanj slobode. Neke od često korištenih varijabli su spomenuta invarijantna masa W i virtualnost Q^2. Uvode se i bezdimenzionalne varijable x_B (Bjorkenov x) i y koje imaju raspon između 0 i 1 te predstavljaju elasticitet, odnosno neelasticitet procesa. Također se koristi i varijabla ν koja u sustavu gdje proton miruje u početnom stanju predstavlja izgubljenu energiju elektrona pri raspršenju.

Poznat je općeniti izraz koji generalizira Rosenbluthovu formulu na neelastični slučaj koja se za $Q^2 >> m_p^2 y^2$ svodi na:

$$\frac{d\sigma}{dx_B dQ^2} \approx \frac{4\pi\alpha^2}{Q^4}\left[(1-y)\frac{F_2(x_B,Q^2)}{x_B} + y^2 F_1(x_B,Q^2)\right], \tag{2.5}$$

gdje se umjesto form faktora pojavljuju *strukturne funkcije* $F_1(x_B,Q^2)$ i $F_2(x_B,Q^2)$, koje zbog ovisnosti o x_B i Q^2 ne mogu biti interpretirane kao Fourierovi transformati distribucija naboja i magnetskog momenta protona.

Rana istraživanja i određivanje strukturnih funkcija iz mjerenih udarnih presjeka pokazala su dva jako bitna svojstva. Prvo je uočeno da ni $F_1(x_B,Q^2)$ ni $F_2(x_B,Q^2)$ gotovo uopće ne ovise o Q^2 što se naziva Bjorkenovo skaliranje i snažno sugerira da se raspršenje događa na točkastim konstituentima unutar protona, inače bi objekt konačne veličine unio Q^2 ovisnost kroz form faktor. To saznanje je indikacija postojanja kvarkova unutar nukleona. Drugo svojstvo, koje je bilo uočeno za virtualnost veću od nekoliko GeV-a, bilo je da strukturne funkcije nisu nezavisne.

Pokazalo se da vrijedi tzv. Callan-Gross relacija:

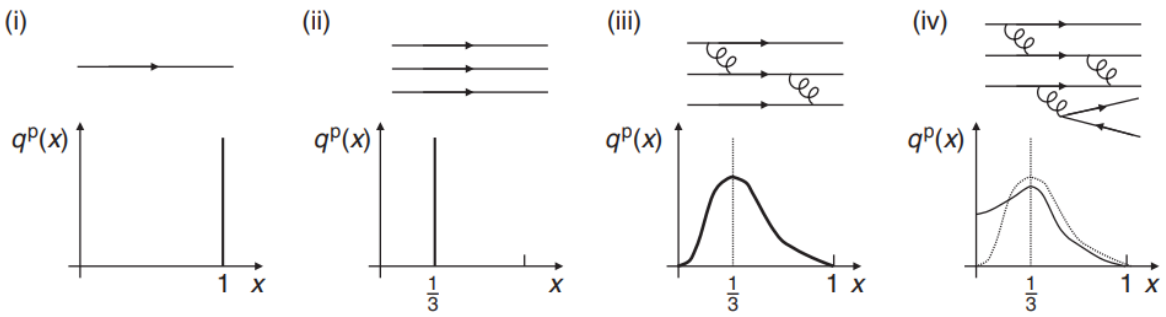
$$F_2(x_B) = 2x_B F_1(x_B), \tag{2.6}$$

koja se može objasniti pod pretpostavkom da je u režimu Bjorkenovog skaliranja pozadinski proces neelastičnog raspršenja zapravo elastično raspršenje elektrona na jednom kvarku koji ima spin 1/2. Prije nego su kvarkovi i gluoni postali generalno prihvaćeni, Feynman je predložio model u kojem je proton bio sazdan od točkastih konstituenata koje je nazvao *partonima* zbog čega i danas gore opisani pozadinski proces nazivamo kvark-partonski model.

Kako strukturne funkcije ne ovise o referentnom sustavu, koristeći sustav beskonačnog impulsa može se pokazati da je udio longitudinalnog impulsa x koji nosi kvark s kojim elektron međudjeluje, jednak Bjorkenovoj varijabli x_B. Slijedi da je udarni presjek kod DIS-a moguće reducirati na sumu udarnih presjeka za raspršenja s pojedinačnim kvarkovima u nukleonu. Strukturne funkcije moguće je izraziti kao:

$$F_2(x_B) = 2x_B F_1(x_B) = x_B \sum_{i=1} Q_i^2 q_i^p(x_B) \tag{2.7}$$

gdje su $q_i^p(x_B)$ takozvane *partonske distribucijske funkcije* (PDF-ovi) koje predstavljaju gustoću partona i u protonu p s frakcijom longitudinalnog momenta x_B u smislu da je broj partona i između x_B i $x_B + \delta x_B$ dan sa $q_i^p(x_B)\delta x_B$. Q_i u ovom slučaju predstavlja naboj partona i u jedinicama naboja protona. Na slici 2.3 vidimo četiri primjera mogućih oblika PDF-ova u protonu.

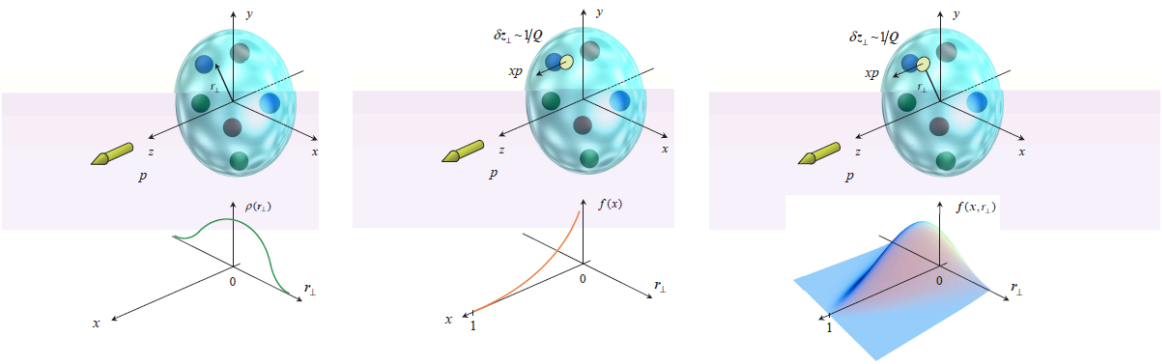


Slika 2.3: (i) jedna točkasta čestica; (ii) tri statična kvarka od kojih svaki nosi po trećinu impulsa (iii) tri kvarka koja interagiraju i izmjenjuju impuls (iv) interagirajući kvarkovi uključujući dijagrame višeg reda. Posuđeno iz [1].

Rano je postalo jasno da je proton puno složeniji nego što se očekivalo i da je slika protona kao vezano stanje tri ,,valentna" kvarka previše pojednostavljena. Osim kvarkova, proton sadrži i ,,more" gluona koji sami nose gotovo polovicu impulsa, a preko produkcije para $g \to q\overline{q}$ unose antikvarkovsku komponentu.

2.3 Generalizirane partonske distribucije

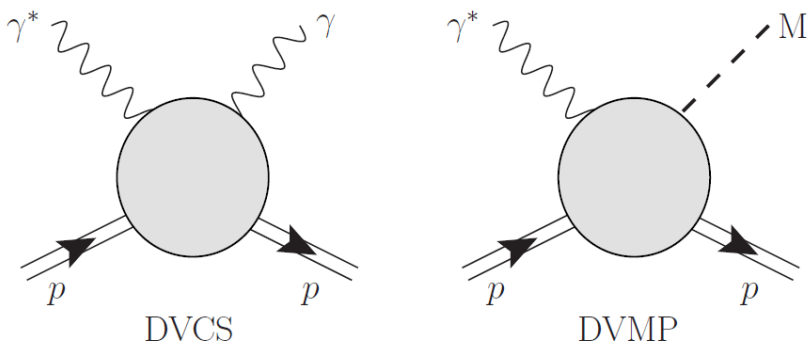
Dok form faktori i partonske distribucije pružaju važne informacije, ni izbliza nisu dovoljne da bismo imali potpuno razumijevanje strukture nukleona. Zato su 90-ih godina prošloga stoljeća uvedene generalizirane partonske distribucije koje su povezane s gore spomenutim objektima. Razlika je u tome što GPD-ovi obuhvaćaju i distribuciju naboja u transverzalnoj ravnini i raspodjelu longitudinalnog impulsa što efektivno daje trodimenzionalnu informaciju (slika 2.4).



Slika 2.4: Probabilistička interpretacija form faktora, partonskih gustoća i generaliziranih partonskih distribucija za $\eta = 0$ u sustavu beskonačnog impulsa. Posuđeno iz [2].

Uz to, GPD-ovi su s čime su ti matrični elementi postali dohvatljivi preko eksperimenata elektromagnetskog raspršenja, umjesto teško izvodljivog gravitacijskog raspršenja.

Teorijski najčišći kanal za pristup GPD-ovima je ekskluzivna elektroprodukcija realnog fotona ili mezona pomoću nukleonske mete pri velikom prijenosu impulsa. Dva procesa koja zadovoljavaju tražene kriterije su duboko virtualno komptonsko raspršenje (DVCS) i duboko virtualna produkcija mezona (DVMP) prikazani na slici 2.5. Pristup GPD-ovima preko navedenih procesa je indirektan jer DVCS i DVMP ne ovise direktno o GPD-ovima, već o njihovim integralima - komptonskim form faktorima (CFF-ovima).



Slika 2.5: Važni procesi za ekstrakciju GPD-ova: lijevo DVCS, desno DVMP. Posuđeno iz [4].

DVCS amplituda može biti izražena kao razvoj produkta operatora. *Twist* je tada definiran kad $d - j$, gdje je d dimenzija tih operatora, a j njihov spin i koristi se za sortiranje članova razvoja po rastućoj potenciji od $1/Q$. Za DVCS, vodeći doprinos je twist-2 što znači da je twist-3 potisnut za $1/Q$, twist-4 za $1/Q^2$ itd.

Zbog kasnijih definicija i izraza bit će nam potrebna sljedeća notacija kao u [4]. Za svaki četverovektor a definiramo *koordinate svjetlosnog stošca* kao:

$$a^{\pm} = \frac{1}{\sqrt{2}}(a^0 \pm a^3) \quad \text{i} \quad a = (a^+, \mathbf{a}, a^-), \tag{2.8}$$

dok se skalarni produkt takva dva vektora definira kao:

$$(ab) = a^+b^- + a^-b^+ - \mathbf{a} \cdot \mathbf{b}. \tag{2.9}$$

Promatrat ćemo hadronske matrične elemente oblika $\langle P_2|O|P_1\rangle$ za različite operatore između ulaznog i izlaznog stanja. Ukupni impuls i prijenos impulsa definiraju se sa:

$$\begin{aligned} P &= P_1 + P_2, \\ \Delta &= P_2 - P_1. \end{aligned} \tag{2.10}$$

Mandelstamova varijabla t odgovara vrijednosti Δ^2, varijabla ξ je kinematička varijabla koja je jednaka $x_B/(2 - x_B)$, γ_μ je Diracova matrica, $\sigma_{\mu\nu} = i[\gamma_\mu, \gamma_\nu]/2$ i $g^{\mu\nu}$ je standardni metrički tenzor. Dodatno uvodimo i GPD varijablu asimetrije (engl. *skewness* kao:

$$\eta = -\frac{\Delta^+}{P^+} \tag{2.11}$$

Općenito, postoje GPD-ovi u nepolariziranom (vektorskom) sektoru i u polariziranom (aksijalno-vektorskom) sektoru. Nepolarizirani GPD-ovi su povezani s prosjecima preko heliciteta kvarkova, dok su polarizirani vezani uz razlike. Također, razlikujemo kiralno-parne GPD-ove koji čuvaju helicitet kvarka na kojem se događa raspršenje od kiralno-neparnih koje isti mijenjaju. DVCS amplituda nije osjetljiva na kiralno-neparne GPD-ove, a gluoni ne doprinose u vodećem redu (twist-2).

Stoga nadalje razmatramo četiri kvarkovska kiralno-parna GPD-a koji parametriziraju strukturu nukleona za twist-2: H_q, E_q, $\tilde{H}_q$ i $\tilde{E}_q$. GPD-ovi ovise o Q^2, x, η i t, ali kao i kod DIS-a, GPD-ovi isto tako pokazuju svojstvo skaliranja i ovisnost o Q^2 je poznata pa se ta ovisnost izbacuje iz notacije i nadalje se podrazumijeva ovisnost o ostale tri spomenute varijable, odnosno $F^q = F^q(x, \eta, t)$, gdje je F^q općenita GPD funkcija.

Sada možemo definirati GPD-ove u nepolariziranom (vektorskom) sektoru kao:

$$\frac{h^+}{P^+}H^q + \frac{e^+}{P^+}E^q = \int \frac{dz^-}{2\pi} e^{ixP^+z^-} \langle P_2|\bar{q}(-z)\gamma^+ q(z)|P_1\rangle \Big|_{z^+=0,\mathbf{z=0}}, \tag{2.12}$$

$$\frac{h^+}{P^+}H^g + \frac{e^+}{P^+}E^g = \frac{4}{P_+}\int \frac{dz^-}{2\pi} e^{ixP^+z^-} \langle P_2|G_a^{+\mu}(-z)G_{a\mu}^{+}(z)|P_1\rangle \Big|_{z^+=0,\mathbf{z=0}}, \tag{2.13}$$

i u polazirizanom (aksijalno-vektorskom) sektoru kao:

$$\frac{\tilde{h}^+}{P^+}\tilde{H}^q + \frac{\tilde{e}^+}{P^+}\tilde{E}^q = \int \frac{dz^-}{2\pi} e^{ixP^+z^-} \langle P_2|\bar{q}(-z)\gamma^+\gamma_5 q(z)|P_1\rangle \Big|_{z^+=0,\mathbf{z=0}}, \tag{2.14}$$

$$\frac{\tilde{h}^+}{P^+}\tilde{H}^g + \frac{\tilde{e}^+}{P^+}\tilde{E}^g = \frac{4}{P_+}\int \frac{dz^-}{2\pi} e^{ixP^+z^-} \langle P_2|G_a^{+\mu}(-z)i\epsilon_{\mu\nu}^{\perp}G_a^{\nu+}(z)|P_1\rangle \Big|_{z^+=0,\mathbf{z=0}}, \tag{2.15}$$

gdje su:

$$h^\mu = \bar{u}(P_2)\gamma^\mu u(P_1); \quad e^\mu = \frac{i\Delta_\mu}{2M}\bar{u}(P_2)\sigma^{\mu\nu}u(P_1), \tag{2.16}$$

$$\tilde{h}^\mu = \bar{u}(P_2)\gamma^\mu\gamma_5 u(P_1); \quad \tilde{e}^\mu = \frac{\Delta^\mu}{2M}\bar{u}(P_2)\gamma_5 u(P_1). \tag{2.17}$$

Spinori su normalizirani tako da vrijedi $\bar{u}(p)\gamma^\mu u(p) = 2p^\mu$.

Za malo jednostavniju interpretaciju, vrijedno je pogledati krajnji slučaj tzv. *forward limit* gdje je $P_1 = P_2$, $t = 0$ kada se neki GPD-ovi svode na PDF-ove:

$$H^q(x,0,0) = \begin{cases} q(x), & x > 0 \\ -\bar{q}(-x), & x < 0 \end{cases} \tag{2.18}$$

$$\tilde{H}^q(x,0,0) = \begin{cases} \Delta q(x), & x > 0 \\ \Delta\bar{q}(-x), & x < 0 \end{cases} \tag{2.19}$$

ili kompaktnije:

$$H^q(x,0,0) = \theta(x)q(x) - \theta(-x)\bar{q}(-x) \tag{2.20}$$

$$\tilde{H}^q(x,0,0) = \theta(x)\Delta q(x) + \theta(-x)\Delta\bar{q}(-x) \tag{2.21}$$

$q(x)(\bar{q}(x))$ i $\Delta q(x)(\Delta\bar{q}(x))$ su respektivno kvarkovski (antikvarkovski) nepolarizirani i polarizirani PDF-ovi za okus q. Za konstantni t, prvi moment GPD-ova daje form faktore, a za $\eta = 0$ GPD-ovi se mogu interpretirati kao amplituda vjerojatnosti pronalaženja kvarka s frakcijom longitudinalnom momenta x na danoj transverzalnoj udaljenosti od središta impulsa nukleona.

GPD-ovi također omogućavaju razotkriti informaciju o strukturi spina nukleona. Kao što je pokazano u [3], spin nukleona moguće je razdvojiti na doprinose kvarkova J_q i gluona J_g:

$$\frac{1}{2} = \sum_q J_q + J_g \tag{2.22}$$

te se kvarkovski doprinosi nadalje mogu razdvojiti na njihov ukupni spin S_q i orbitalni angularni moment L_q:

$$J_q = \frac{1}{2}S_q + L_q. \tag{2.23}$$

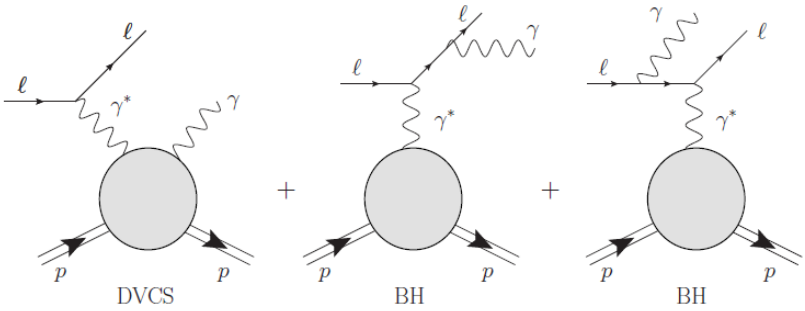
Drugi moment GPD-ova tada daje poznato *Jijevo sumacijsko pravilo*, koje omogućava pristup kvarkovskom doprinosu J_q, a s tim i ukupnom orbitalnom angularnom momentu kvarkova L_q:

$$J_q = \frac{1}{2}\int_{-1}^{1} x[H_q(x,\eta,t=0) + E_q(x,\eta,t=0)]dx. \tag{2.24}$$

U ostala matematička svojstva GPD-ova kao što su pravila sume, pozitivnost i polinomijalnost nećemo detaljnije ulaziti, a opisana su u [4].

2.4 Komptonski form faktori

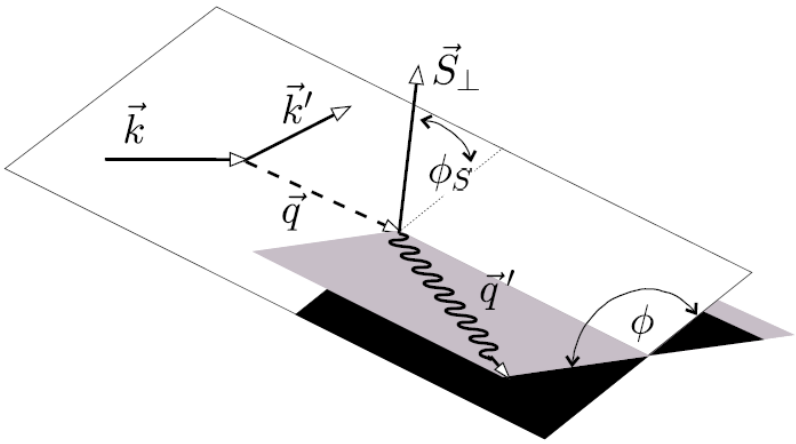
Mjerenja DVCS-a najčešće se izvode preko leptoprodukcije realnog fotona, a pojavljuje se i interferencija s takozvanim Bethe-Heitler procesom što je prikazano na slici 2.6.



Slika 2.6: Leptoprodukcija realnog fotona kao koherentna superpozicija DVCS i Bethe-Heitler amplituda. Posuđeno iz [4].

Eksperimentalno nije moguće razdvojiti DVCS od Bethe-Heitler amplitude. Obje imaju isto početno i konačno stanje, dok u BH slučaju foton biva zračen iz elektrona umjesto kvarka na kojem se elektron raspršuje. Posljedično, doprinosi BH amplitude i njihova interferencija s DVCS trebaju biti uključeni u udarni presjek mjerenja ekskluzivne elektro-produkcije fotona.

Udarni presjek diferencijalan je po x_B, virtualnosti Q, kvadriranom prijenosu četveroimpulsa $t = \Delta^2$ i dva azimutalna kuta mjerena relativno prema leptonskoj ravnini raspršenja. ϕ je kut do ravnine foton-meta raspršenja, a ϕ_S do transverzalne komponente polarizacijskog vektora mete kao što je prikazano na slici 2.7 koja je označena prema Trento konvenciji.



Slika 2.7: Definicija impulsa i kuteva relevantnih za leptoprodukciju realnog fotona. Posuđeno iz [4].

Izraz za udarni presjek dan je sa:

$$\frac{d^5\sigma}{dx_B dQ^2 d|t| d\phi d\phi_S} = \frac{\alpha^3 x_B}{16\pi^2 Q^4 \sqrt{1+\epsilon^2}} \left|\mathcal{T}\right|^2, \tag{2.25}$$

gdje su α elektromagnetska konstanta fine strukture, $\epsilon = 2x_B M/Q$, M je masa mete i $\mathcal{T}$ je koherentna superpozicija DVCS i Bethe-Heitler amplituda:

$$|\mathcal{T}|^2 = |\mathcal{T}_{BH} + \mathcal{T}_{DVCS}|^2 = |\mathcal{T}_{BH}|^2 + |\mathcal{T}_{DVCS}|^2 + \mathcal{I}. \tag{2.26}$$

$\mathcal{I}$ predstavlja interferencijski član, a DVCS amplituda $\mathcal{T}_{DVCS}$ se može raspisati preko amplituda heliciteta ili ekvivalentno preko kompleksnih *komptonskih form faktora* (CFF-ova). Razmatramo četiri CFF-a u najnižem redu: $\mathcal{H}$, $\mathcal{E}$, $\tilde{\mathcal{H}}$ i $\tilde{\mathcal{E}}$ koje je pomoću QCD faktorizacijskih teorema moguće izraziti u vodećem redu perturbativne QCD pomoću sljedećih konvolucija:

$$\mathcal{F}(\eta, t) = \sum_q e_q^2 \int_{-1}^{1} dx \left[\frac{1}{\eta - x - \imath\epsilon} - \frac{1}{\eta + x - \imath\epsilon}\right] F^q(x, \eta, t), \tag{2.27}$$

$$\tilde{\mathcal{F}}(\eta, t) = \sum_q e_q^2 \int_{-1}^{1} dx \left[\frac{1}{\eta - x - \imath\epsilon} + \frac{1}{\eta + x - \imath\epsilon}\right] \tilde{F}^q(x, \eta, t), \tag{2.28}$$

gdje je $\mathcal{F} = \mathcal{H}, \mathcal{E}, ...$ CFF funkcije te $F^q = H^q, E^q, ...$ pripadajuće GPD funkcije i odakle vidimo da su CFF jednostavnije jer ovise samo o dvije varijable koje određuju kinematičko područje te funkcije. Bethe-Heitler amplituda izražena je pomoću elastičnih form faktora koji se u kinematičkoj regiji eksperimenata poznati do na 1% što skupa s interferencijskim članom daje eksperimentalni pristup realnim i imaginarnim dijelovima CFF-ova. Moguće je pokazati da su konačni izrazi [4] za amplitude koje ulaze u udarni presjek dane sa:

$$|\mathcal{T}_{BH}|^2 = \frac{1}{x_B^2 t (1+\epsilon^2)^2 \mathcal{P}_1(\phi)\mathcal{P}_2(\phi)} \left\{ c_0^{BH} + \sum_{n=1}^{2} c_n^{BH} cos(n\phi) + s_1^{BH} sin\phi \right\}, \tag{2.29}$$

$$|\mathcal{T}_{DVCS}|^2 = \frac{1}{Q^2} \left\{ c_0^{DVCS} + \sum_{n=1}^{2} \left[c_n^{DVCS} cos(n\phi) + s_n^{DVCS}\right] \right\}, \tag{2.30}$$

$$\mathcal{I} = \frac{-e_l}{x_B t y \mathcal{P}_1(\phi)\mathcal{P}_2(\phi)} \left\{ c_0^{\mathcal{I}} + \sum_{n=1}^{3} \left[c_n^{\mathcal{I}} cos(n\phi) + s_n^{\mathcal{I}} sin(n\phi)\right] \right\}, \tag{2.31}$$

gdje je y gubitak energije leptona u referentnom sustavu mete, e_l naboj snopa leptona u jedinicama naboja pozitrona, a $1/(\mathcal{P}_1(\phi)\mathcal{P}_2(\phi))$ dolazi od leptonskih propagatora u BH amplitudi.

CFF-ovi ulaze kvadratično u harmoničke koeficijente c_n^{DVCS} i s_n^{DVCS} iz DVCS amplitude, linearno u one iz interferencijskog člana te su potpuno odsutni iz BH dijela izraza. Točni izrazi za koeficijente mogu se naći u [5].

2.5 DVCS opservable

Kako bismo iz pokazanih izraza došli do CFF funkcija, potrebna su DVCS mjerenja udarnih presjeka. Uvelike bi pomoglo kada bi na raspolaganju imali snopove leptona različitih naboja (elektron i pozitron) i spinova, kao i mogućnost različite polarizacije mete. Također bi pomoglo kada bi za metu mogli koristiti i proton i neutron. Na taj način, dobile bi se različite opservable kojima različiti CFF-ovi različito doprinose. Neke od tih pretpostavki nažalost još nije moguće eksperimentalno ostvariti sa željenom preciznošću.

Udarni presjek za leptoprodukciju realnog fotona leptonom l (s nabojem e_l u jedinicama naboja pozitrona i helicitetom $h_l/2$) koji se raspršuje na nepolariziranoj meti može se zapisati kao:

$$d\sigma^{h_l,e_l}(\phi) = d\sigma_{UU}(\phi)[1 + h_l A_{LU,DVCS}(\phi) + e_l h_l A_{LU,I} + e_l A_C(\phi)], \tag{2.32}$$

gdje je samo ovisnost o kutu ϕ prikazana eksplicitno. Najčešće se koristi notacija u kojoj prvi indeks označava polarizaciju snopa: „U" znači nepolarizirano (engl. *unpolarized*), „L" označava longitudinalnu polarizaciju, dok „T" predstavlja transverzalnu polarizaciju. Drugi indeks je rezerviran za polarizaciju mete.

Primjerice, ako eksperimentalno postoje uvjeti za longitudinalnu polarizaciju snopa leptona i dostupni su i pozitivno i negativno nabijeni snopovi, moguće je izolirati *asimetrije*:

$$A_C(\phi) = \frac{(d\sigma^{+}_{\rightarrow} + d\sigma^{+}_{\leftarrow}) - (d\sigma^{-}_{\rightarrow} + d\sigma^{-}_{\leftarrow})}{4d\sigma_{UU}(\phi)}, \tag{2.33}$$

$$A_{LU,I}(\phi) = \frac{(d\sigma^{+}_{\rightarrow} - d\sigma^{+}_{\leftarrow}) - (d\sigma^{-}_{\rightarrow} - d\sigma^{-}_{\leftarrow})}{4d\sigma_{UU}(\phi)}, \tag{2.34}$$

$$A_{LU,DVCS}(\phi) = \frac{(d\sigma^{+}_{\rightarrow} - d\sigma^{+}_{\leftarrow}) + (d\sigma^{-}_{\rightarrow} - d\sigma^{-}_{\leftarrow})}{4d\sigma_{UU}(\phi)} \tag{2.35}$$

Koristi se uobičajena notacija gdje $\rightarrow$ i $\leftarrow$ označavaju desni odnosno lijevi helicitet, a $+$ i $-$ pozitivan ili negativan naboj leptonskog snopa. A_C predstavlja asimetriju naboja snopa (engl. *beam charge asymmetry*, BCA) što se vidi i iz izraza, dok $A_{LU,I}$ i $A_{LU,DVCS}$ predstavljaju asimetrije spina snopa (engl. *beam spin asymmetry*, BSA).

S druge strane, ako eksperiment ima pristup samo jednoj vrijednosti e_l kao što je to slučaj na Jefferson Lab-u, prethodno definirane asimetrije ne mogu biti izolirane i može se mjeriti asimetrija spina snopa $A^{e_l}_{LU}$ koja ovisi o kombiniranom naboj-spin udarnom presjeku:

$$A^{e_l}_{LU}(\phi) = \frac{d\sigma^{e_l}_{\rightarrow} - d\sigma^{e_l}_{\leftarrow}}{d\sigma^{e_l}_{\rightarrow} + d\sigma^{e_l}_{\leftarrow}} \tag{2.36}$$

Moguće je i mjeriti asimetriju spina mete (engl. *target spin asymmetry*, TSA), kao i dvostruku asimetriju spina (engl. *beam target spin asymmetry*, BTSA) koje se definiraju analogno uz dodatak da $\Leftarrow$ ($\Rightarrow$) signalizira paralelnu ili antiparalelnu polarizaciju mete u odnosu na impuls snopa.

Kod eksperimenata koji ne mogu mjeriti direktne udarne presjeke, već samo asimetrije, može se iskoristiti činjenica da je u nazivniku dominantan BH član tako da se dobije približno linearna ovisnost opservable o CFF-ovima kao što je to slučaj za prvi sinusni harmonik asimetrije spina snopa s longitudinalno polariziranim snopom:

$$A^{-,\sin\phi}_{LU}(\phi) \equiv \int_{-\pi}^{\pi} d\phi \sin\phi A^{-}_{LU}(\phi). \tag{2.37}$$

Taj harmonik je onda približno proporcionalan linearnoj kombinaciji CFF-ova:

$$A^{-,\sin\phi}_{LU}(\phi) \propto Im\left(F_1\mathcal{H} - \frac{t}{4M^2}F_2\mathcal{E} + \frac{x_B}{2}(F_1+F_2)\tilde{\mathcal{H}}\right) \tag{2.38}$$

i tim izrazom dominira Im$\mathcal{H}$. Ako se mjere udarni presjeci, često se vrši Fourierova analiza ili se radi s posebnim težinskim Fourierovim mjerama kako bi se poništili oscilirajući faktori $1/(\mathcal{P}_1(\phi)\mathcal{P}_2(\phi))$ u Bethe-Heitler i interferencijskim članovima. Redovi takvih težinskih harmoničkih članova konvergiraju brže nego standardni Fourierov red.

Konačno, u DVCS mjerenjima pojavljuju se *polarizirani* i *nepolarizirani* udarni presjeci. Polarizirani udarni presjek (engl. *beam spin difference*, BSD) definiran je kao razlika udarnih presjeka s različitim spinom odnosno helicitetom:

$$\Delta\sigma = \frac{1}{2}(d\sigma^{\rightarrow} - d\sigma^{\leftarrow}), \tag{2.39}$$

dok je nepolarizirani udarni presjek (engl. *beam spin sum*, BSS) se definira kao zbroj udarnih presjeka s lijevim i desnim helicitetom:

$$d\sigma = d\sigma^{\rightarrow} + d\sigma^{\leftarrow}. \tag{2.40}$$

Postoji više metoda izvlačenja (ekstrakcije) nepoznatih CFF-ova iz navedenih opservabli. Neke od njih su jako primitivne i nepouzdane kao što je tzv. lokalna ekstrakcija gdje se odrede numeričke vrijednosti nekih od CFF-ova za fiksne parametre η i t. U drugoj varijanti lokalne ekstrakcije, teorijski se odrede minimumi i maksimumi vrijednosti svih CFF-ova što dovodi do konvergencije i relativno preciznih vrijednosti za $Im\mathcal{H}$ koji je dominantni doprinos za većinu opservabli, dok su ostale vrijednosti relativno nepouzdane.

Nama je interesantno dobiti globalnu ekstrakciju nasuprot lokalne. Jedna od opcija je pomoću teorijskih saznanja pretpostaviti funkcijski oblik za svaki od CFF-ova sa po nekoliko slobodnih parametara koje naučimo prilagodbom na mjerenja opservabli, za sve η i t istovremeno. To funkcionira dobro, no postavlja se pitanje pristranosti koja se unosi u model samim odabirom funkcijskih oblika. Tim odabirom značajno smanjujemo funkcijski prostor jer funkcija ovisi o dvije kinematičke varijable na relativno nepoznat način, a za konačni cilj (poznavanje GPD-ova) taj bi problem postao još i veći. Upravo je to motivacija zbog koje za ovu svrhu pribjegavamo neuronskim mrežama koje parametriziraju proizvoljne CFF funkcije bez da unose pristranost.

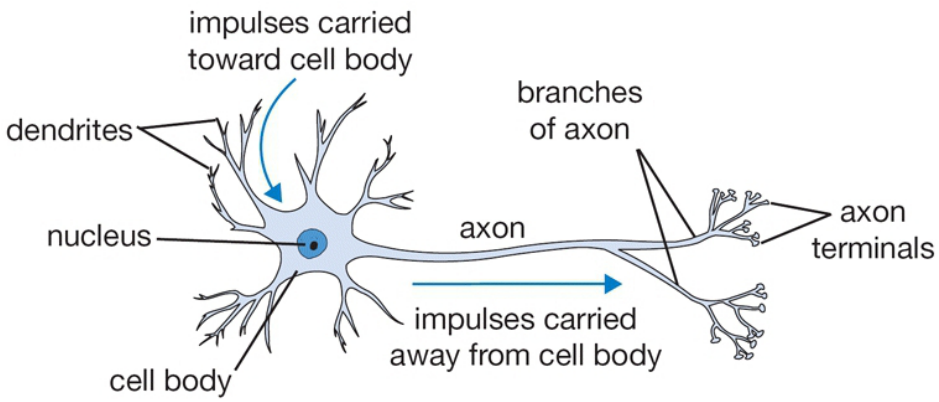
3 Umjetne neuronske mreže

3.1 Motivacija i povijesni uvod

Još od kraja 19. stoljeća, postoji teorijski okvir na osnovu kojega su nastale današnje neuronske mreže (Bain i James). Njihove hipoteze povezivale su ljudske misli i tjelesnu aktivnost s interakcijama neurona u mozgu. U jednoj verziji različite aktivnosti vode na pobuđenje različitih neurona i tako jačaju veze među njima, dok u drugoj akcije nastaju kao rezultat električne struje kroz neurone što ne zahtijeva vezu među svim pojedinačnim neuronima.

Pojam umjetna neuronska mreža (engl. *artificial neural network*, ANN) potječe od pokušaja pronalaska matematičke reprezentacije obrade informacija u biološkim sustavima (McCulloch i Pitts 1943.) pa se tako i danas koristi za široki spektar algoritama za prepoznavanje uzoraka (npr. prepoznavanje znamenki, lica, objekata i sl.) modeliranih po uzoru na ljudski mozak.

Naime, biološki neuroni prikazani na slici 3.1. su fundamentalne jedinice mozga i živčanog sustava odgovorne za primanje senzorskih informacija iz okoline preko dendrita, a potom i za obradu istih te izlaz preko aksona.



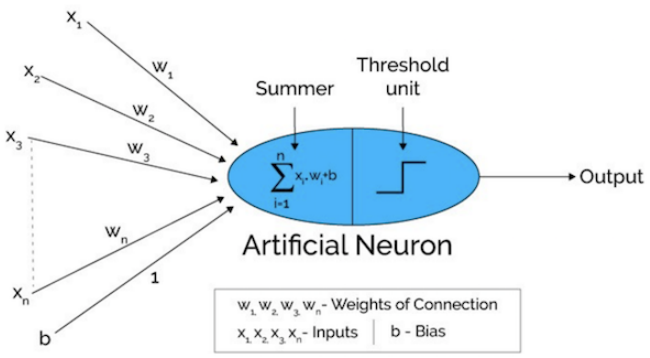
Slika 3.1: Biološki neuron. Posuđeno iz [6].

Početni korak u simuliranju načina rada prikazanog biološkog neurona bila je pojava algoritma *perceptrona* (Rosenblatt, 1958.), koji se naziva i umjetnim neuronom, a koristi se za binarnu klasifikaciju. Istraživanja u ovom području smanjuju svoj intenzitet krajem 60-tih godina 20. stoljeća, dijelom zbog nedostatka računalne snage, dijelom zbog nedostatka efikasnog algoritma za učenje modela te zbog nemogućnosti perceptrona da riješi linearno neodvojiv problem (npr. *exclusive-or*). Situacija se znatno mijenja krajem 80-tih godina pojavom algoritma propagacije unatrag (engl.

backpropagation), što omogućava razmjernu prilagodbu onih dijelova mreže koji su odgovorni za pogrešku na izlazu, otkada područje neuro-računarstva doživljava rapidan napredak i široku primjenu zbog čega veza s biološkim kontekstom ostaje samo na razini motivacije.

3.2 Umjetni neuron - perceptron

Kao što je spomenuto, bitan korak dogodio se s modelom perceptrona (umjetnog neurona), koji je prikazan na slici 3.2.



Slika 3.2: Perceptron. Posuđeno iz [7].

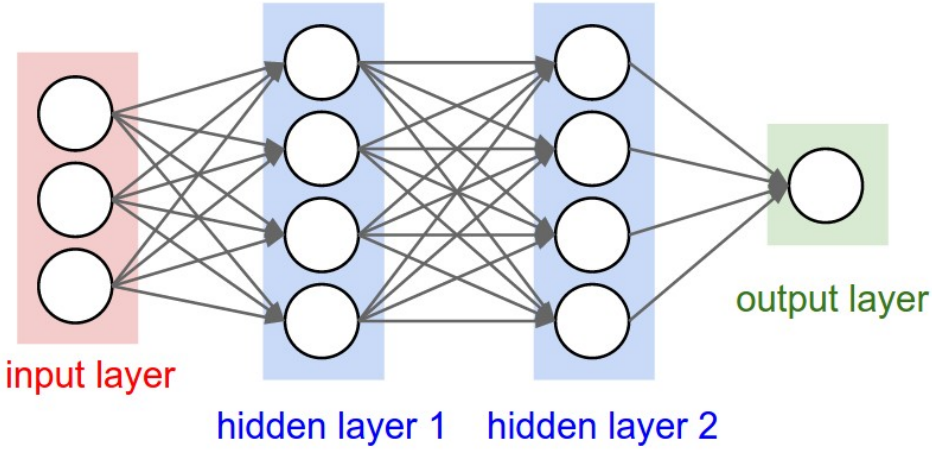
S gornje slike vide se glavni njegovi elementi. Ulazi x_0, x_1, ... i x_n predstavljaju značajke jednog primjera, dok je b označava pristranost (engl. *bias)* što je parametar koji služi da decizijska granica u prostoru značajki ne prolazi nužno kroz ishodište. Svaka od tih značajki ulazi u model s jednom od $n+1$ težina $w_o = 1$, w_1, ..., w_n koje su proporcionalne važnosti pripadne značajke:

$$a = b + \sum_{j=1}^{n} w_j \cdot x_j \tag{3.1}$$

Djelovanjem aktivacijske funkcije na skalarni produkt značajki s težinama dobijamo predikciju za dani primjer kao $\hat{y} = f(a)$. Aktivacijska funkcija f kod perceptrona je step funkcija θ, zbog čega je perceptron linearan model. Optimizacija se vrši tako da se predikcija primjera uspoređuje s danim oznakama skupa za učenje i težine se ažuriraju ako oznaka ne odgovara predikciji sve dok se ne nađe vektor težina za koji su sve predikcije jednake stvarnim oznakama primjera.

3.3 Neuronske mreže

Za neuronske mreže se približno može reći da su višeslojni perceptroni. Umjetni neuroni (čvorovi) kombiniraju se s pripadnim težinama u više čvorova u takozvanom skrivenom sloju (engl. *hidden layer*) koji sada služi kao ulaz za sljedeći sloj tako stvarajući umjetnu neuronsku mrežu. Način na koji su neuroni međusobno organizirani i povezani u mreži određuje njezinu *arhitekturu*. Među ostalima, razlikujemo acikličku (engl. *feedforward*), mrežu s povratnom vezom (engl. *recurrent net*), lateralno povezanu i hibridnu neuronsku mrežu. Među acikličkim se ističu potpuno povezana mreža(slika 3.3.), konvolucijska mreža i autoenkoder. Kao broj slojeva mreže, najčešće se uzima njen broj slojeva bez ulaznog sloja.

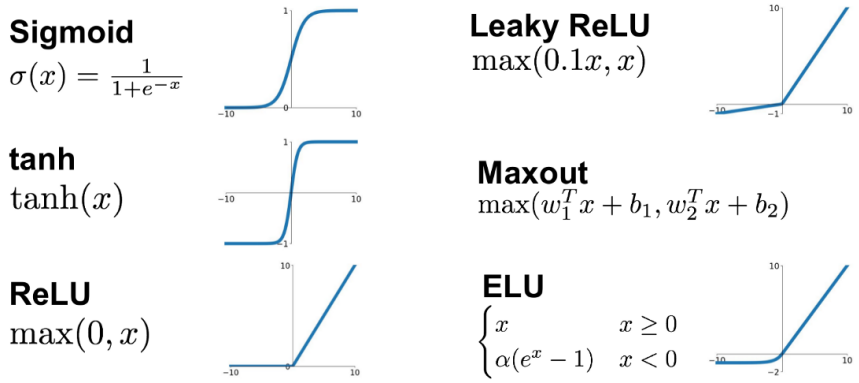


Slika 3.3: Primjer neuronske mreže sa 3 sloja. Posuđeno iz [6].

Arhitektura ovisi o zadanom problemu i području u kojem se algoritam primjenjuje (npr. duboko učenje, obrada prirodnog jezika ili računalni vid), dok će se u sklopu ovog rada koristiti samo aciklička potpuno povezana mreža.

Bitna razlika između višeslojnog perceptrona i neuronske mreže je u aktivacijskoj funkciji. Kod perceptrona, kao što je spomenuto, radi se o step funkciji koja odlučuje je li neuron ,,aktiviran" ili ne. Kada bi imali više slojeva, izlaz bi i dalje bio linearna kombinacija ulaza. Takav model ima jako malenu složenost i nema kapacitet naučiti i modelirati složnije nelinearne podatke.

Zbog toga se u neuronskim mrežama koriste nelinearne aktivacijske funkcije koje se primjenjuju na skrivene slojeve. Na taj način, neuronske mreže mogu aproksimirati bilo koju funkciju kao što ćemo kasnije ilustrirati. Neke od nelinearnih aktivacijskih funkcija kao što su sigmoida ili logistička funkcija, tangens hiperbolni i ReLu (*rectified linear unit*) su prikazane na slici 3.4.



Slika 3.4: Nelinearne aktivacijske funkcije. Preuzeto iz [8].

U općenitom slučaju, neka neuronska mreža ima L slojeva i neka l-ti sloj ima n_l čvorova, gdje $l = 1, 2, ..., L$. Neka je n broj značajki, odnosno dimenzija ulaznih vektora tako da je $\mathbf{x} = (x_1, x_2, ..., x_n)^T$. Za k-tu aktivaciju u prvom sloju vrijedi:

$$a_k^{(1)} = \sum_{i=1}^{n} w_{ki}^{(1)} x_i + b_k^{(1)}, \tag{3.2}$$

gdje su $w_{ki}^{(1)}$ i $b_k^{(1)}$ težine i pristranosti s kojima čvorovi „nultog" sloja (ulazne značajke) ulaze u k-ti čvor prvog sloja mreže. Svaka aktivacija transformira se koristeći spomenute diferencijabilne, nelinearne aktivacijske funkcije $h(\cdot)$ i daje takozvana skrivena stanja (engl. *hidden units*):

$$z_k^{(1)} = h(a_k^{(1)}) = h\left(\sum_{i=1}^{n} w_{ki}^{(1)} x_i + b_k^{(1)}\right), \tag{3.3}$$

gdje navedeno vrijedi za svaki k od 1 do n_1, $\mathbf{a^{(1)}} = (a_1^{(1)}, a_2^{(1)}, ..., a_{n_1}^{(1)})^T$ i $\mathbf{z^{(1)}} = (z_1^{(1)}, z_2^{(1)}, ..., z_{n_1}^{(1)})^T$. Prema obliku jednadžbi i poopćenjem na bilo koji sloj, parametri l-tog sloja mogu se kompaktno zapisati u obliku matrice težina $W^{(l)}$ koja ima dimenzije $(n_l \times n_{l-1})$ i vektora pristranosti $\mathbf{b^{(l)}}$ dimenzije $(n_l \times 1)$:

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1,n_{l-1}}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2,n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l,1}^{(l)} & w_{n_l,2}^{(l)} & \cdots & w_{n_l,n_{l-1}}^{(l)} \end{pmatrix} \quad i \quad \mathbf{b^{(l)}} = (b_1^{(l)}, b_2^{(l)}, ..., b_{n_l}^{(l)})^T. \tag{3.4}$$

l-ti sloj tada ima $(n_l + 1) \cdot n_{l-1}$ parametara.

Za prvi sloj onda možemo pisati:

$$\mathbf{z}^{(\mathbf{1})} = h\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(\mathbf{1})}\right), \tag{3.5}$$

za l-ti sloj možemo pisati:

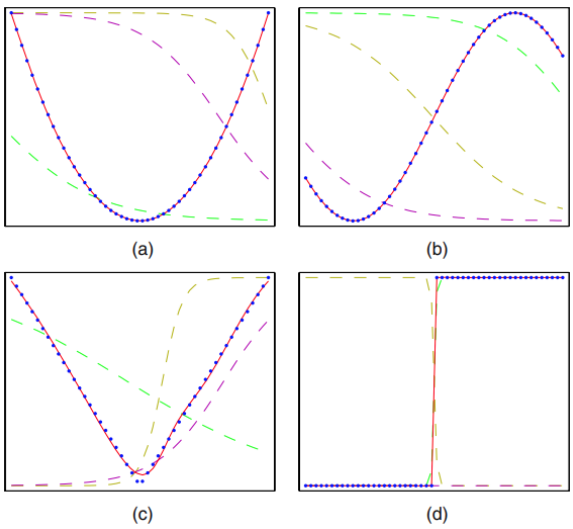
$$\mathbf{z}^{(\mathbf{l})} = h\left(W^{(l)}\mathbf{z}^{(\mathbf{l-1})} + \mathbf{b}^{(\mathbf{l})}\right), \tag{3.6}$$

dok izlazni vektor iz mreže (predikciju) označavamo sa $\hat{\mathbf{y}}$ i računamo kao:

$$\hat{\mathbf{y}} = h\left(W^{(L)}\mathbf{z}^{(\mathbf{L-1})} + \mathbf{b}^{(\mathbf{L})}\right). \tag{3.7}$$

3.4 *Aproksimacijska svojstva neuronskih mreža*

Aproksimacijska svojstva kod acikličkih (engl. *feed-forward*) neuronskih mreža bila su predmet mnogih istraživanja (npr. [9]). Pokazano je da su ta svojstva jako općenita zbog čega se neuronske mreže nazivaju *univerzalnim aproksimatorima*. Primjerice, dvoslojna mreža s linearnim izlazima može aproksimirati bilo koju kontinuiranu funkciju na kompaktnoj domeni s proizvoljnom preciznošću, uz uvjet da mreža ima dovoljan broj čvorova u skrivenom sloju. Taj rezultat vrijedi za različite aktivacijske funkcije i prikazan je na slici 3.5 na problemu regresije.



Slika 3.5: Ilustracija sposobnosti dvoslojne neuronske mreže. Preuzeto iz [10].

Kod regresije je ciljna vrijednost y kontinuirana te je iz skupa primjera $\mathcal{D} = \left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)$ potrebno naučiti $f : \chi \to \mathbb{R}$ tako da u idealnom slučaju vrijedi $\mathbf{y}^{(i)} = \hat{\mathbf{y}} = f(\mathbf{x}^{(i)})$ za svaki i.

Na slici se vide četiri funkcije, čijih je 50 točaka označenih plavim točkama nasumično izvučeno na domeni $[-1, 1]$, aproksimirane neuronskom mrežom što je prikazano crvenom linijom. Funkcije su (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c) $f(x) = |x|$ i (d) $f(x) = \theta(x)$, gdje je $\theta(x)$ Heaviside step funkcija. Neuronska mreža ima tri čvora u skrivenom sloju čiji su doprinosi na slikama prikazani crtkanim linijama, odakle se vidi kako individualni čvorovi iz skrivenog sloja rade zajedno kako bi što bliže opisali ciljnu funkciju.

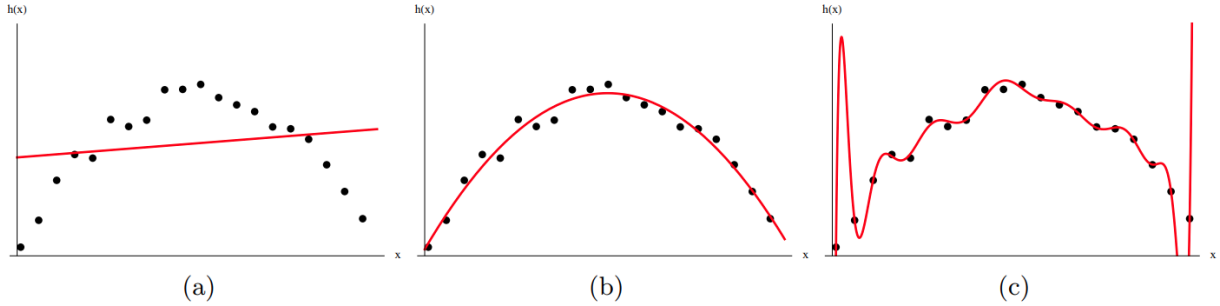
3.5 Odabir modela

Činjenica da su neuronske mreže univerzalni aproksimatori pod nekim pretpostavkama nam pomaže, no ostaje nam problem kako a priori znati je li model dovoljno složen i ima li dovoljan kapacitet za opisati traženi problem. Učenje hipoteze interesantan je problem utoliko što je riječ o loše definiranom problemu (engl. *ill-posed problem*), odnosno primjeri za učenje $\mathcal{D}$ nisu sami po sebi dovoljni da bi se na temelju njih jednoznačno inducirala hipoteza h za koju želimo da ima ključno svojstvo *generalizacije*: predviđanje klase ili vrijednosti dotad neviđenih primjera.

To sugerira uvođenje *induktivne pristranosti*, dodatnih pretpostavki koje omogućavaju induktivno učenje s kojima se odlučujemo za neki skup hipoteza (model) što se dalje svodi na optimizaciju *hiperparametara*. Zbog mnogih razloga (računalni resursi, interpretabilnost) preferiramo što jednostavnije modele, a da je istovremeno sposobnost generalizacije što veća. Kako bismo na neki način mjerili sposobnost generalizacije, potreban nam je kriterij tj. funkcija empirijske pogreške hipoteze koja se za regresiju najčešće definira kao:

$$E(h|\mathcal{D}) = \frac{1}{2}\sum_{i=1}^{N} \|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|^2, \tag{3.8}$$

gdje koristimo vektorsku notaciju, $\hat{\mathbf{y}}^{(i)}$ je predikcija neuronske mreže za ulazni vektor $\mathbf{x}^{(i)}$, dok je 1/2 uvršten zbog jednostavnosti optimizacije. Kod problema regresije u stvarnosti, zbog neizostavne prisutnosti šuma koji nastaje iz različitih razloga, učimo funkciju $f(\mathbf{x}^{(i)}) + \epsilon$, gdje je ϵ slučajni šum. Naš je cilj optimizacijom parametara mreže (što ćemo detaljnije obraditi naknadno), minimizirati empirijsku pogrešku, opisati podatke i istovremeno moći generalizirati. Kada dobijemo naučen model, razlikujemo 3 situacije prikazane na slici 3.6.



Slika 3.6: Regresija funkcije $f(x) = -x^2 + \epsilon$. : (a) podnaučenost (linearna regresija), (b) optimalan model (regresija polinomom drugog stupnja), (c) prenaučenost (regresija polinomom 15. stupnja). Preuzeto iz [11].

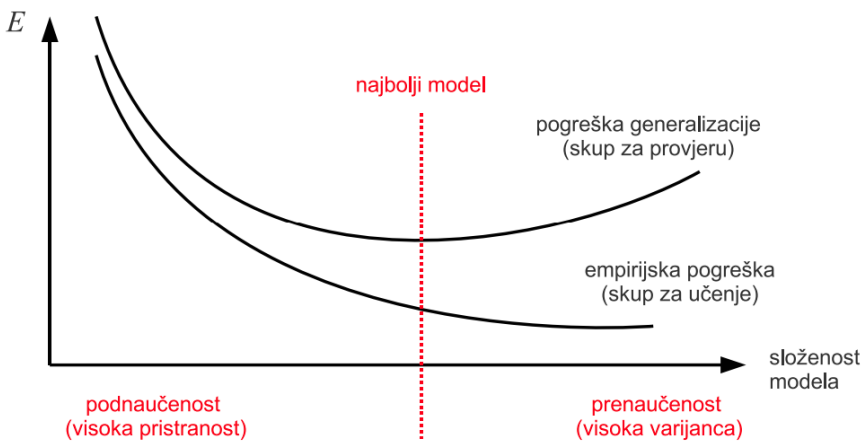
Prvi dio slike pokazuje situaciju podnaučenosti (engl. *underfitting*) što odgovara prejednostavnom modelu (veća statistička pristranost) i manjku kapaciteta modela da se prilagodi podacima pa je jako izgledno da će takav model biti loš pri generalizaciji neviđenih primjera.

Treći dio slike pokazuje primjer prenaučenosti (engl. *overfitting*) što odgovara presloženom modelu koji pretpostavlja više od onoga što se nalazi u podacima te se zapravo prilagođava šumu u podacima. Takvi modeli imaju veću varijancu što znači da će male promjene u skupu podataka dovesti do većih promjena u predikcijama.

U ovom slučaju optimalan model je polinom drugog stupnja koji predstavlja idealan balans između pristranosti i varijance (engl. *bias-variance dilemma*).

Jednostavan način da se kvantitativno izabere optimalna složenost modela jest unakrsna provjera (engl. *cross-validation*). Tada podatke razdvajamo na dva djela: skup za učenje (engl. *training set*) i skup za provjeru (engl. *validation set*). Model se uči na skupu za učenje, a njegovu generalizacijsku sposobnost provjeravamo na njemu disjunktnom skupu za provjeru. Na taj način, može se vrlo dobro procijeniti kako se model ponaša na neviđenim primjerima. Pogrešku hipoteze mjerenu na skupu koji nije korišten za učenje nazivamo pogreškom generalizacije. Tipično ponašanje empirijske pogreške i pogreške generalizacije prikazano je na slici 3.7.

U režimu podnaučenosti obje pogreške su velike, dok kod prenaučenosti vidimo da je pogreška na skupu za učenje puno manje od pogreške generalizacije.



Slika 3.7: Empirijska pogreška i pogreška generalizacije u ovisnosti o složenosti modela. Preuzeto iz [11].

3.6 Optimizacija parametara i gradijentni spust

Kako bi neuronska mreža što bolje obavila svoj zadatak, u našem slučaju što vjernije aproksimirala funkciju koju opisuju podaci, potrebno je provesti optimizaciju njenih parametara. Taj zadatak se svodi na nalaženje matrica težina i vektora pristranosti za koje je funkcija pogreške minimalna. U daljnjem tekstu zanemarujemo ovisnost o pristranostima b jer se za njih može provesti analogan postupak.

U slučaju kada se ne može dobiti analitičko rješenje jednadžbe minimuma $\nabla E(\mathbf{w}) = 0$, pribjegavamo raznim iterativnim algoritmima optimizacije. Nakon inicijalizacije težina i početnog prolaska kroz mrežu računa se pogreška te se na osnovu toga težine ažuriraju u svrhu smanjivanja pogreške. Pravilo ažuriranja općenito je oblika:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}. \tag{3.9}$$

Koristeći činjenicu da je gradijent vektor u smjeru najvećeg porasta funkcije, najjednostavnije i najkorištenije pravilo ažurira težine u smjeru negativnog gradijenta tako smanjujući grešku prema minimumu:

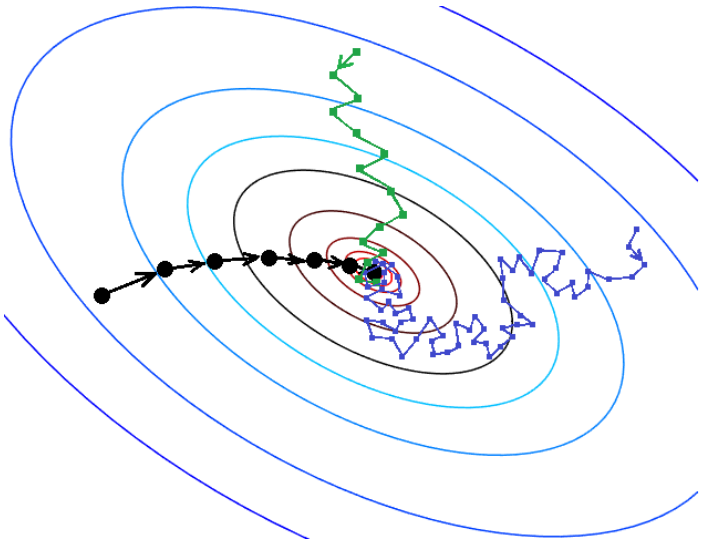
$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E(w^{(\tau)}), \tag{3.10}$$

gdje je η hiperparametar koji najčešće odabiremo unakrsnom provjerom i nazivamo stopa učenja (engl. *learning rate*). Ako je η prevelik ažuriranjem težina preskočimo minimum, a ako je premalen učenje može trajati predugo. Samo pravilo ažuriranja naziva se gradijentni spust (engl. *gradient descent*).

Postoje tri varijante gradijentnog spusta ovisno o broju primjera iz skupa koje uzimamo u obzir pri računanju pogreške, a potom i gradijenata. Prva opcija je takozvani *batch* gradijentni spust u kojemu se računa prosječna pogreška na cijelom skupu za učenje i ažuriranje težina se vrši nakon svake epohe odnosno nakon svakog prolaska cijelog skupa za učenje unaprijed. Druga opcija je *mini-batch* gradijentni spust gdje se skup svih primjera za učenje dijeli na disjunktne podskupove odabrane veličine. Tada se pogreška, a onda i gradijenti, računaju na svakom podskupu (mini-batchu) pojedinačno te se ažuriranja težina vrše nakon što svaki mini-batch prođe unaprijed i unatrag kroz mrežu. Treći i krajnji slučaj je stohastički gradijentni spust (engl. *online learning*) gdje se funkcija gubitka L definira kao funkcija pogreške na jednom primjeru:

$$\mathcal{L}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) = \frac{1}{2}\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|^2. \tag{3.11}$$

Tada se težine ažuriraju nakon prolaska svakog primjera kroz mrežu. Za očekivati je da će takav postupak rezultirati ne nužno monotonim padom funkcije gubitka uprosječene po jednoj epohi i skokovitijim ponašanjem. Tu činjenicu možemo iskoristiti da objasnimo mogućnost stohastičkog gradijentnog spusta da lakše ,,pobjegne" iz lokalnog minimuma jer stacionarna točka funkcije pogreške na cijelom skupu podataka generalno nije jednaka stacionarnoj točki za svaku točku pojedinačno. Konvergencija ovih triju slučajeva prikazana je na slici 3.8.



Slika 3.8: Vrste gradijentnog spusta. Crna linija predstavlja *batch* gradijentni spust, plava stohastički gradijentni spust,a zelena *mini-batch* gradijentni spust. Preuzeto iz [12].

Batch gradijentni spust računa stvarni gradijent, ali je najsporiji. Stohastički gradijentni spust računa grubu procjenu gradijenta, ali je najbrži. Mini-batch varijanta je kompromis koji koristi prednosti obje krajnosti.

Važno je napomenuti da u višedimenzionalnim prostorima parametara (duboko učenje) imamo puno više problema sa sedlenim točkama nego lokalnim minimumima. Da bi točka bila minimum, bilo globalni bilo lokalni, ta točka mora biti minimum te funkcije po svakoj od njenih n komponenti. Vjerojatnost da se to dogodi je $1/2^n$ što je za veliki n očigledno zanemarivo. S druge strane sedlena točka je po nekim komponentama minimum, dok je po nekima maksimum.

3.7 Algoritam propagacije unatrag

Preostaje još dati objašnjenje kako većina današnjih neuronskih mreža koristi pravilo ulančanih derivacija kako bi ubrzala proces učenja i učinila ga efikasnijim. Radi se u algoritmu propagacije unatrag (engl. *backpropagation algorithm*) [13], koji je istraživanje neuronskih mreža ponovno stavio u fokus krajem 80-tih godina prošlog stoljeća.

Za gradijentni spust i mnoge druge optimizacijske postupke ažuriranja težina potrebno je izračunati derivacije funkcije gubitka s obzirom na sve težine. Neka je $\mathcal{L}$ korištena funkcija gubitka, $\mathbf{a}^{(\mathbf{l})}$ su aktivacije, a $\mathbf{z}^{(\mathbf{l})}$ su skrivena stanja u l-tom sloju. Također koristeći prethodnu notaciju neka je h aktivacijska funkcija koja je jednaka za sve slojeve (radi pojednostavljenja izvoda, lako je poopćiti), L je ukupan broj slojeva mreže i $W^{(l)} = w_{ij}^{(l)}$ su težine koje pripadaju l-tom sloju.

Za zadnji sloj koristeći pravilo ulančanog deriviranja možemo pisati:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(L)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(L)}} \cdot \frac{\partial z_i^{(L)}}{\partial a_i^{(L)}} \cdot \frac{\partial a_i^{(L)}}{\partial w_{ij}^{(L)}}. \tag{3.12}$$

Znamo da je $\mathbf{z}^{(\mathbf{l})} = h(\mathbf{a}^{(\mathbf{l})})$ i $a_i^{(l)} = \sum_{k=1} w_{ik}^{(l)} z_k^{(l-1)} + b_i^{(l)}$ za svaki l između 1 i L iz čega slijedi da je:

$$\frac{\partial \mathcal{L}}{\partial z_i^{(L)}} = \mathcal{L}'(z_i^{(L)}) \ , \quad \frac{\partial z_i^{(L)}}{\partial a_i^{(L)}} = h'(a_i^{(L)}) \quad i \quad \frac{\partial a_i^{(L)}}{\partial w_{ij}^{(L)}} = z_j^{(L-1)}. \tag{3.13}$$

Tada imamo:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(L)}} = \mathcal{L}'(z_i^{(L)}) \cdot h'(a_i^{(L)}) \cdot z_j^{(L-1)} \ , \tag{3.14}$$

gdje su $\mathcal{L}'$ i h' poznate derivacije funkcije pogreške i aktivacijske funkcije, a $z_j^{(L-1)}$ aktivacija predzadnjeg sloja.

Kada se ne radi o zadnjem sloju nego nekom unutarnjem sloju l, gdje je $l < L$, situacija je drugačija utoliko što tada prvi faktor s desne strane jednadžbe za neuron iz sloja l analogne jednadžbi (3.12), derivacija funkcije gubitka $\mathcal{L}$ po skrivenim stanjima l-tog sloja $z_i^{(l)}$, ovisi o svakom neuronu koji dobiva ulaz iz razmatranog neurona.
Tada imamo:

$$\frac{\partial \mathcal{L}}{\partial z_i^{(l)}} = \sum_{k=1}^{n_{l+1}} \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}. \tag{3.15}$$

Iz $z_i^{(l)} = h\left(\sum_{k=1} w_{ik}^{(l)} z_k^{(l-1)} + b_i^{(l)}\right)$ slijedi:

$$\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = h'(a_k^{(l+1)}) \cdot w_{ki}^{(l+1)} \tag{3.16}$$

pa jednadžba (3.15) postaje:

$$\frac{\partial \mathcal{L}}{\partial z_i^{(l)}} = \sum_{k=1}^{n_{l+1}} \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \cdot h'(a_k^{(l+1)}) \cdot w_{ki}^{(l+1)} \tag{3.17}$$

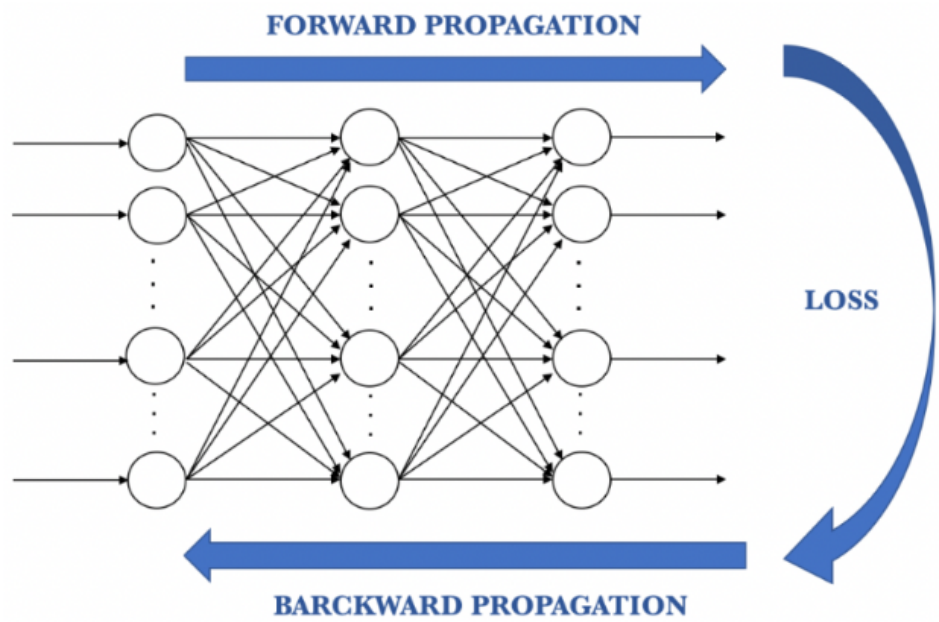
Drugi faktor je analogan prethodnom slučaju, derivacija skrivenog stanja po pripadnoj aktivaciji je i dalje jednaka derivaciji aktivacijske funkcije za vrijednost te aktivacije. Treći faktor također ostaje neovisan o drugim dijelovima mreže. u svrhu kompaktnijeg zapisa uvodimo standardnu pokratu $\delta^{(l)}$ kao:

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial a_i^{(l)}} = h'(a_k^{(l)}) \times \begin{cases} \mathcal{L}'(z_i^{(l)}), & l = L \\ \sum_{k=1}^{n_{l+1}} w_{ki}^{(l+1)} \delta_k^{(l+1)}, & l < L \end{cases} \tag{3.18}$$

Tada konačno objedinjeno možemo zapisati:

$$\Delta w_{ij}^{(l)} = \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \cdot z_j^{(l-1)} \tag{3.19}$$

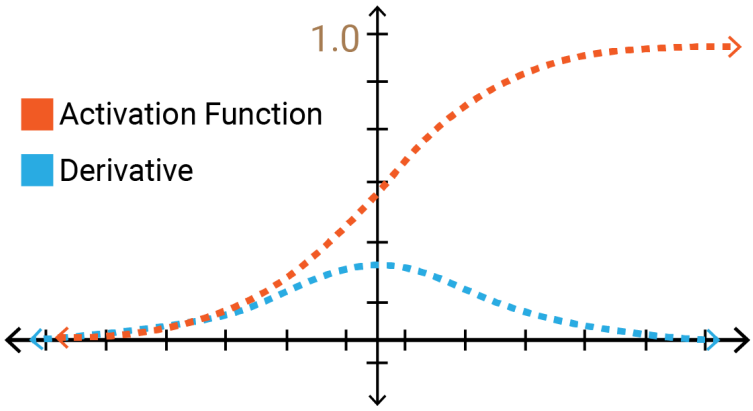
Vidimo da je varijabla $\delta^{(l)}$ određena rekurzivnom jednadžbom. Znamo da se prolaskom mrežom unaprijed računaju sve aktivacije i skrivena stanja, a iz jednadžbe (3.18) se lako zaključi da jednom kad izračunamo $\delta^{(L)}$, znamo i $\delta^{(L-1)}$. Kada znamo $\delta^{(L-1)}$, lako se izračuna i $\delta^{(L-2)}$ i tako dalje sve do $\delta^{(1)}$. U toj spoznaji leži snaga i efikasnost algoritma propagacije unatrag. Krećući od posljednjeg sloja, pri računanju gradijenata bliže ulaznom sloju već imamo izračunat i dostupan dobar dio informacije koja nam je potrebna. Jedan ciklus učenja mreže u 2 koraka, propagacije unaprijed i unatrag, prikazan je na slici 3.9.



Slika 3.9: Ciklus učenja neuronske mreže. Preuzeto iz [14].

3.8 Problemi pri učenju dubokih mreža

Kako ćemo u ovom radu isprobavati i neke arhitekture koje upadaju u režim dubokog učenja, susrest ćemo se i s nekim karakterističnim problemima. Najučestaliji je problem iščezavajućih gradijenata (engl. *vanishing gradient problem*), koji se najčešće vezuje uz aktivacijske funkcije kao što su sigmoid ili tangens hiperbolni. Takvim funkcijama je derivacija efektivno različita od nule samo u relativno uskom pojasu oko nule što se vidi na slici 3.10. Iz opisanog algoritma propagacije unatrag vidi se da promjena težina, a time i gradijenti u ranijim slojevima, ovisi o umnošku gradijenata iz slojeva koji dolaze nakon njih.



Slika 3.10: Sigmoidna aktivacijska funkcija i njena derivacija. Preuzeto iz [15].

Stoga je jasno da se u arhitekturama s velikim brojem slojeva lako dogodi da gradijenti u početnim slojevima duboke mreže iščeznu i da se efektivno ne događa učenje kroz mrežu. Najčešće su početni dijelovi mreže jako važni u raspoznavanju značajki iz podataka pa ovaj problem dovodi do velikih nepreciznosti.

Taj problem djelomično se rješava koristeći spomenutu ReLu aktivacijsku funkciju koja ima konstantnu derivaciju jednaku jedan na pozitivnom dijelu svoje domene, ali s druge strane može uzrokovati takozvano odumiranje neurona. Drugi način su rezidualne mreže koje imaju direktne veze između kasnijih i ranijih slojeva, a u koje nećemo detaljnije ulaziti.

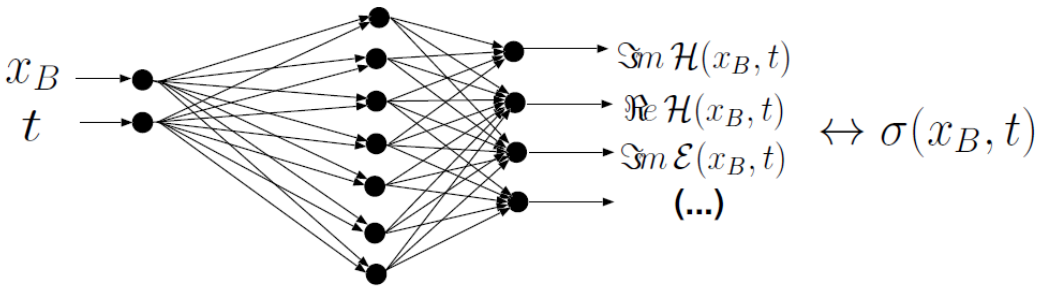
Može se dogoditi i eksplozija gradijenata (engl. *exploding gradient problem*) kada se gradijenti funkcije gubitka akumuliraju i čine učenje nestabilnim. Najčešće se rješava ograničenjem gradijenta (engl. *gradient clipping*) gdje se gradijent ,,odreže" na određenoj vrijednosti koju unaprijed određujemo.

Valja napomenuti i važnost inicijalizacije težina. Ako se radi o arhitekturi s većim brojem slojeva i čvorova po sloju, može se dogoditi da varijanca izlaza iz sloja bude puno veća od varijance ulaza. To opet dovodi do regije gdje su derivacije blizu nuli, odnosno do prestanka učenja. Za svaki tip aktivacijske funkcije postoji preporučena vrsta inicijalizacije težina koja pomaže pri očuvanju varijance pri prolasku kroz mrežu. Kod sigmoide i tangensa hiperbolnog to je takozvana ,,Xavier" inicijalizacija [16], dok uz ReLu funkciju najbolje radi ,,He" inicijalizacija [17].

4 Ekstrakcija CFF funkcija

4.1 Umjetni podaci

Već smo spomenuli da bi neuronske mreže mogle biti dobar izbor za modeliranje komptonskih form faktora zato što zahvaljujući svojim svojstvima ne unose pristranost u model. Ta mogućnost testirana je u [18] uz prilagodbu mentorovog „GeParD" softvera na *Tensorflow*, a nezavisno potvrđena u [19]. Pomoću poznatog teorijskog Goloskokov-Kroll (GK) modela [20] za CFF-ove i poznatih formula kreirane su vrijednosti za razne opservable koristeći kinematičke točke (ξ, t) s mjerenja na CLAS detektoru. Kako bi se što vjernije simulirala mjerenja sa stvarnih eksperimenata, na tako dobivene vrijednosti opservabli dodan je slučajni šum što zajedno daje set umjetnih (engl. *mock*) podataka. Tada su CFF funkcije $Im\mathcal{H}$, $Re\mathcal{H}$ i $Im\tilde{\mathcal{H}}$ modelirane neuronskim mrežama na istoj kinematičkoj domeni, a njihov izlaz korišten je za predikciju opservabli. Analogna postavka prikazana je na slici 4.1.



Slika 4.1: Neuronska mreža kao set CFF-ova. Mreža se uči računajući udarne presjeke (opservable) pomoću CFF-ova, uspoređujući ih s eksperimentalnim mjerenjima i korigirajući parametre mreže u cilju smanjenja pogreške. Preuzeto iz [21].

Cilj je bio naučiti generirane umjetne podatke, koji su bili korišteni za računanje pogreške koja se propagirala natrag kroz mrežu i korigirala se na način da predikcije opservabli budu što bliže vrijednostima umjetnih podataka. Kao indirektan rezultat tog postupka, neuronske mreže tada trebaju dati funkcije jako bliske funkcijama iz GK modela. Rezultati su bili zanimljivi i vizualno zadovoljavajući. Uspješnost ekstrakcije CFF funkcija mjerena je prosjekom korijena kvadratne pogreške (engl. *root mean squared error*, RMSE) te su dobivene i kombinacije opservabli koje su zajedno pogodne za ekstrakciju. To je predstavljalo dobru validaciju neuronske mreže kao metode jer u slučaju kad se bude radilo sa stvarnim podacima, znamo da metoda rezultira s ekstrakcijom funkcija približnih inherentnim funkcijama koje generiraju mjerene opservable.

U ovom radu, razmatra se procedura slična opisanoj, ali s korištenjem stvarnih umjesto umjetnih podaraka. Koriste se mjerenja s eksperimenata CLAS (opservable $BSDw$, $BSSw0$, $BSSw1$, $BTSA$, TSA i BSA), Hall A(opservable $BSDw$, $BSSw0$, $BSSw1$) i HERMES (opservabla BCA).

4.2 Izbor arhitekture i značajki mreže

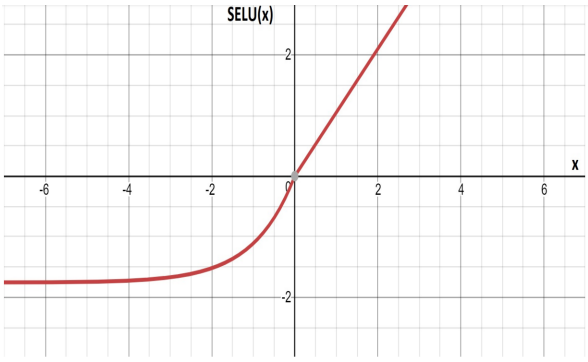
Prvi dio provedenih eksperimenata odnosi se na izbor arhitekture i značajki neuronske mreže kako bi se dobili što bolji rezultati. Koristeći i ažurirajući mentorov kod te prilagodbu istoga na *Tensorflow* [18], testirane su neuronske mreže s različitim brojem slojeva (1, 2, 10, 50, 100) i različitim brojem čvorova po sloju (10, 50, 100, 200, 500, 1000). U ovom dijelu za predikciju opservabli korišteno je 6 CFF funkcija ($Im\mathcal{H}$, $Re\mathcal{H}$, $Im\tilde{\mathcal{H}}$, $Im\mathcal{E}$, $Re\mathcal{E}$ i $Im\tilde{\mathcal{E}}$) kao u [22]. Sve CFF funkcije modelirane su istom mrežom sa zajedničkim parametrima koja u zadnjem sloju ima broj čvorova jednak broju CFF-ova, svaki čvor predstavlja vrijednost po jedne CFF funkcije koje dalje ulaze u račun udarnog presjeka.

U svakom modelu je korišten poznati Adam optimizacijski algoritam [23], koji koristi prvi i drugi moment gradijenata parametara i na taj način „izglađuje" nagle skokove u gradijentima i čini optimizacijski postupak robusnijim i primjenjivim na široki raspon problema.

Stopa učenja je hiperparametar koji se bira unakrsnom provjerom i blago se smanjuje s brojem epoha što dovodi do manjih oscilacija stohastičkog spusta oko minimuma i efikasnije konvergencije. Aktivacijska funkcija SELU (engl. *Scaled Exponential Linear Unit*) koja je korištena u svim slojevima prikazana je na slici 4.2, a definirana je kao:

$$SELU(x) = \begin{cases} \lambda x, & x > 0 \\ \lambda\alpha(e^x - 1), & x < 0 \end{cases} \tag{4.1}$$

Vidi se da ova funkcija koristi prednost ReLu funkcije (dovoljno velika derivacija u pozitivnom dijelu domene da se izbjegne iščezavanje gradijenata) i rješava se njene mane (sprječava pojavu odumiranja neurona u negativnom dijelu domene). U ovome dijelu rada korišteni su samo podaci s CLAS detektora (180 točaka) koji su podijeljeni na skup za učenje i skup za validaciju na isti način za svaku postavku mreže zbog reproducibilnosti rezultata.

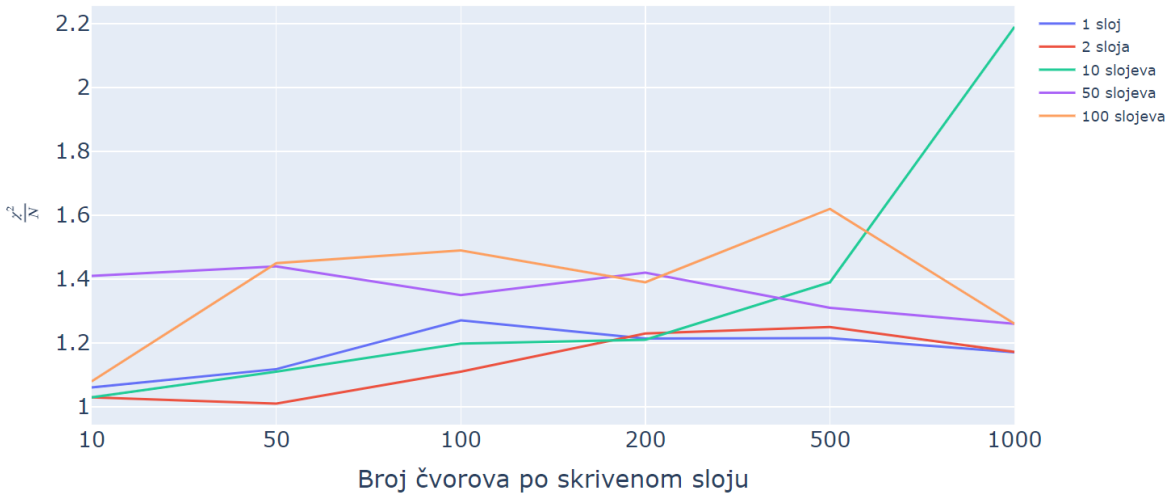


Slika 4.2: SELU(x) za $\alpha = 1.673$ i $\lambda = 1.051$. Preuzeto iz [24].

Kvaliteta prilagodbe neuronske mreže na mjerene podatke opisuje se χ^2 testom:

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{\Delta y_i^2}, \tag{4.2}$$

gdje je y_i mjerenje opservable s neodređenosti Δy_i, predikcija neuronske mreže za danu točku označena je sa $\hat{y}_i$, a N je broj mjerenih točaka. χ^2/N bi idealno trebao biti oko 1, no prihvatljive su i nešto veće vrijednosti. Vrijednosti za χ^2/N na rešetci ispitivanih arhitektura (broj slojeva $\times$ broj čvorova po sloju) dane su na slici 4.3.



Slika 4.3: Vrijednosti χ^2/N za ispitivane arhitekture.

Podaci su podijeljeni na skup za učenje i skup za validaciju u omjeru 3:1, a prikazane vrijednosti odnose se na oba skupa kombinirano. Dobiveni rezultati prikazani su i u tablici 4.1. gdje se za mrežu od 2 i 10 slojeva vide i vrijednosti χ^2/N na skupu za validaciju u zagradama. Idealno bismo željeli da su χ^2/N vrijednosti slične i na skupu za učenje i na skupu za validaciju jer veliki nesrazmjer može ukazivati na prenaučenost.

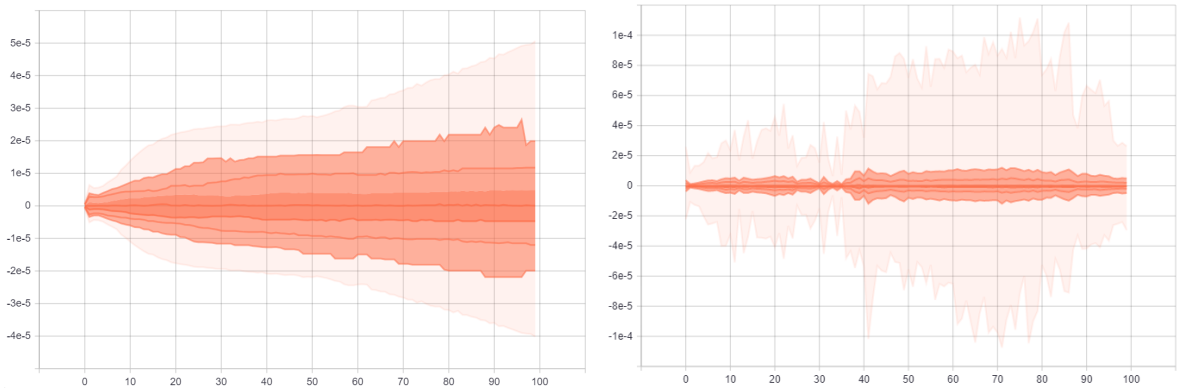
Broj slojeva	1	2	10	50	100
10 čvorova po sloju	1.06	1.03 (1.46)	1.03 (1.34)	1.41	1.08
50 čvorova po sloju	1.12	**1.01** (1.42)	1.11 (**1.29**)	1.44	1.45
100 čvorova po sloju	1.27	1.11 (1.54)	1.20 (1.50)	1.35	1.49
200 čvorova po sloju	1.21	1.23 (1.58)	1.21 (1.67)	1.42	1.39
500 čvorova po sloju	1.22	1.25 (1.82)	1.39 (1.82)	1.31	1.62
1000 čvorova po sloju	1.17	1.17 (1.50)	2.19 (2.97)	1.26	1.26

Tablica 4.1: Vrijednosti χ^2/N za ispitivane arhitekture. U zagradama se nalaze vrijednosti izračunate na skupu za validaciju.

U našim eksperimentima to ne mora biti slučaj jer su izvršeni samo za jednu od mnogih mogućih konfiguracija raspodjele primjera u dva skupa. Vidi se da konfiguracija mreže (2×50) daje najbolju ukupnu vrijednost (1.01). Međutim, mreža (10×50) daje nižu vrijednost na skupu za validaciju (1.29). Ovi rezultati djelomično mogu biti posljedica većeg broja netipičnih vrijednosti (engl. *outliera*) u skupu za validaciju, a bolja prilagodba mreže na podatke iz skupa za validaciju jedne arhitekture u odnosu na drugu bi mogla ukazivati na bolju sposobnost generalizacije. Ponovno zbog činjenice da je eksperiment izvršen samo na jednoj konfiguraciji podjele podataka to može biti istina samo za tu konfiguraciju primjera, ne nužno za sve ostale moguće načine podjele. Zbog činjenice da je svaki od ovih 30 eksperimenata trajao otprilike 3 sata, ograničeni smo s mogućnostima pokušaja iste procedure na više različitih podjela podataka u skupove. Zato se okrećemo drugom načinu za donošenje informirane odluke o optimalnoj arhitekturi.

Koristimo *Tensorboard*, alat koji služi za vizualizaciju Tensorflow grafa koji izvršava sve operacije, pomoću kojega je implementirano i omogućeno praćenje procesa učenja na način da vidimo parametre mreže i njihove gradijente za svaki sloj u obliku distribucije i histograma. Uspoređujući gradijente kroz mrežu za dvije diskutirane situacije, mreže (2×50) i (10×50) prikazane na slici 4.4, vidimo da su gradijenti u prvom sloju mreže (2×50) nešto šire raspodijeljeni nego oni u šestom sloju mreže (10×50).

To implicira smanjenu mogućnost učenja mreže (10×50) i situaciju u kojem se zadnjih nekoliko slojeva adaptira na podatke koristeći informacije s ulaza u puno manjoj mjeri zbog nemogućnosti propagacije pogreške unazad i manjeg ažuriranja parametara.

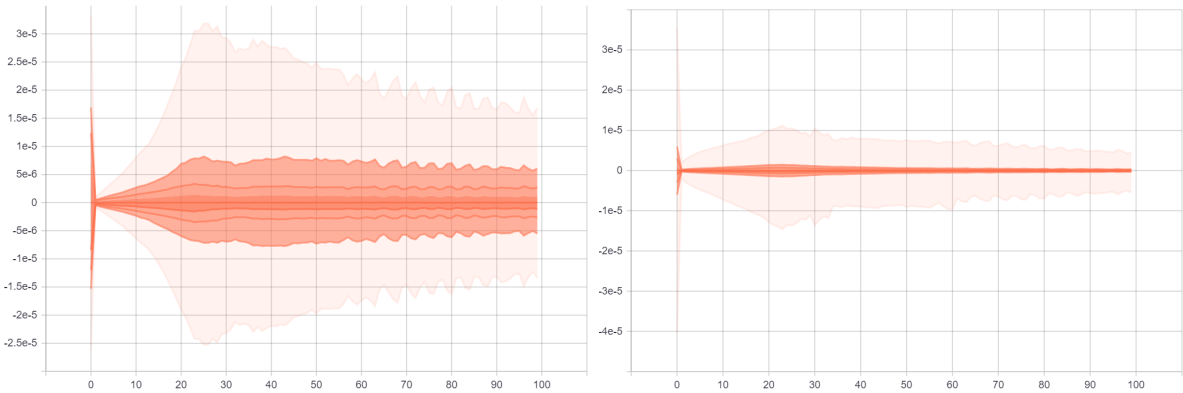


(a) (b)

Slika 4.4: Distribucija gradijenata za a) 1. sloj (2×50) mreže i b) 6. sloj (10×50) mreže

Na slikama nijanse od intenzivnijih prema manje intenzivnim označavaju redom područja ograničena s prosjekom, jednom standardnom devijacijom, dvije standardne devijacije, tri standardne devijacije i minimuma ili maksimuma matrica gradijenata. Vrijednosti gradijenata prikazane su preko 100 epoha kroz koliko je proces učenja izveden.

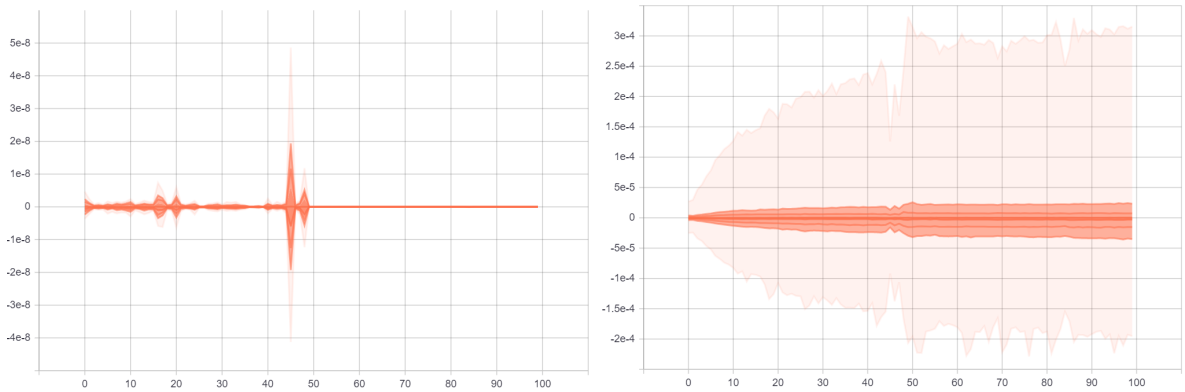
Na isti način možemo promotriti i drugačije konfiguracije. Tako primjerice slučaj mreža s 2 sloja s 1000 čvorova po sloju, prikazan na slici 4.5, unatoč sličnim χ^2/N vrijednostima kao (2×50) slučaj, pokazuje najmanje dvostruko užu distribuciju gradijenata u prvom sloju i gotovo zanemarive gradijente u svom drugom sloju. To čini mreže s 2 sloja i puno čvorova po sloju nepotrebnima te ih isključujemo iz razmatranja jer unatoč tome što imaju više parametara, ne daju nikakvo poboljšanje performansama modela.



(a) (b)

Slika 4.5: Distribucija gradijenata za a) 1. sloj (2×1000) i b) 2. sloj (2×1000) mreže

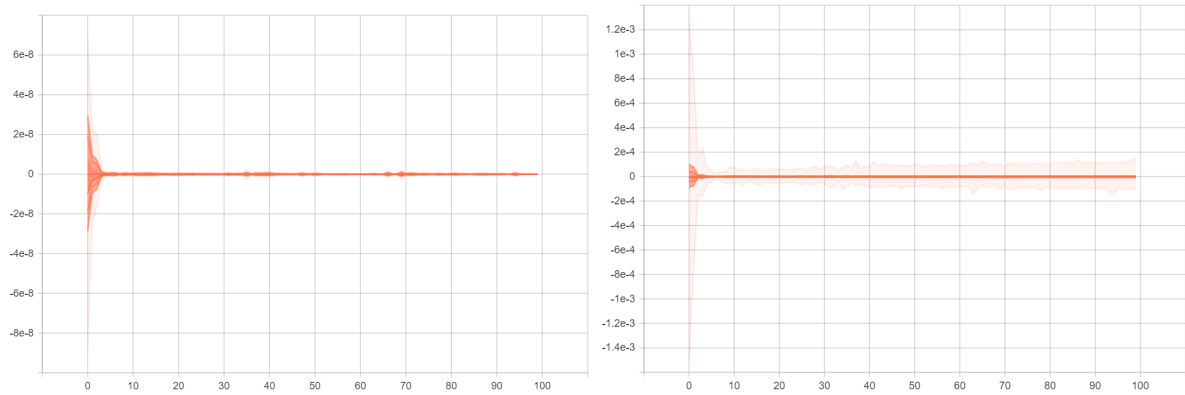
S druge strane, promotrimo i slučaj s više slojeva kao što je mreža (100×100). Ovdje se primjećuje klasična situacija iščezavajućih gradijenata. Širina distribucije gradijenata smanjuje se od zadnjeg prema prvom sloju na način da je svaki sloj prije 90. praktički zanemariv, odnosno zaustavlja učenje i propagaciju pogreške unatrag. Na slici 4.6 prikazane su distribucije gradijenata za 1. i 100. sloj (100×100) mreže.



(a) (b)

Slika 4.6: Distribucija gradijenata za a) 1. sloj i b) 100. sloj (100×100) mreže

Ponavlja se situacija da se zadnjih nekoliko slojeva donekle prihvatljivo prilagodi podacima (χ^2/N=1.49), ali kvalitativno se ne dobije dobar opis podataka i sposobnost generalizacije je jako loša. Slična situacija primjećuje se i za mrežu (50×500) koja kvantitativno još bolje opisuje podatke ($\chi^2/N = 1.31$) i prikazana je na slici 4.7.

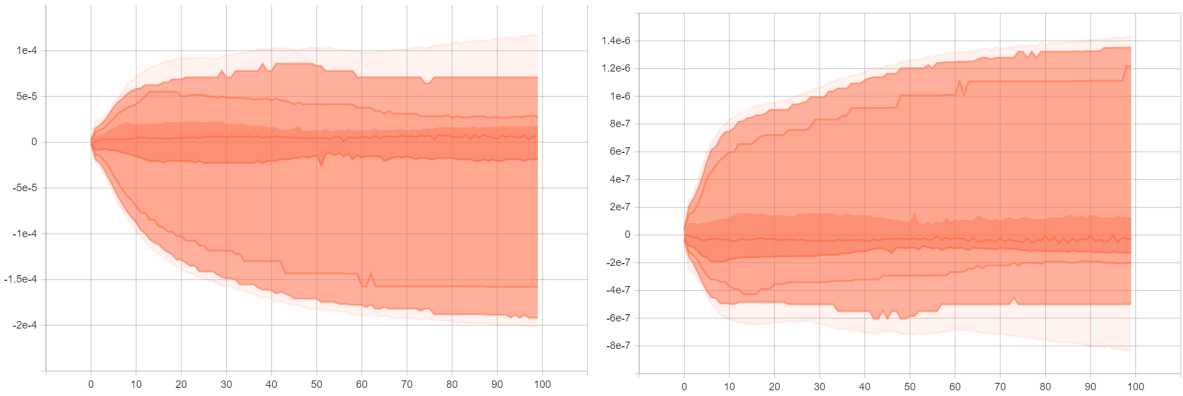


(a) (b)

Slika 4.7: Distribucija gradijenata za a) 1. sloj i b) 50. sloj (50×500) mreže

Sve skupa, nameće se zaključak da je (2×50) optimalna arhitektura neuronske mreže među razmatranima. Također se pokazalo da nema potrebe za velikim brojem čvorova u mreži od dva sloja, kao ni za „dubokim” mrežama s većim brojem slojeva. Iz tog razloga, nadalje se podrazumijeva da svaka korištena mreža ima (2×50) arhitekturu što je sličan raspon kao što je predloženo u [22].

Kao što je kratko spomenuto, umjesto dosadašnjih modela u kojem je svaka CFF funkcija imala svoju neuronsku mrežu sa svojim parametrima, implementirana je i varijanta u kojoj sve promatrane CFF funkcije dijele parametre sve do zadnjeg sloja koji ima onoliko čvorova koliko CFF-ova promatramo (u ovom slučaju 6). Svaki taj čvor tada predstavlja vrijednost određene CFF funkcije koju dalje koristimo za predikciju opservabli. U ovom dijelu analize, to nam pomaže da usporedimo koliko je teško ekstrahirati koju CFF funkciju, odnosno koliko su one dostupne u odnosu na druge. To možemo vidjeti sa slike 4.8.



(a) (b)

Slika 4.8: Distribucija gradijenata za posljednji linearni sloj prema izlazu koji odgovara a) $Im\mathcal{H}$ funkciji i b) $Im\tilde{\mathcal{E}}$ funkciji

S y-osi obiju slika vidimo razliku od nekoliko redova veličine u distribuciji gradijenata što implicira da $Im\mathcal{H}$ ima veći utjecaj na opservable. U tom dijelu mreže se događaju veće korekcije koje poboljšavaju prilagodbu mreže na podatke, odnosno može se reći da je $Im\mathcal{H}$ dostupnija za ekstrakciju od $Im\tilde{\mathcal{E}}$.

Implementirana je i opcija za izbor korisnika želi li provesti normalizaciju ulaza ili regularizaciju što je pokazano u kodu u dodacima. Kada se redovi veličina ulaznih značajki razlikuju, to vodi na otežanu optimizaciju pri čemu normalizacija ulaza pomaže tako da svaka značajka ima varijancu jednaku 1 i prosječnu vrijednost jednaku 0. S druge strane, regularizacija je tehnika koja u funkciju gubitka dodaje član jednak zbroju normi svih težina pomnožen s hiperparametrom koji se bira unakrsnom provjerom. Na taj se način sprječava prenaučenost jer se smanjuju norme težinama, nekima i do nule, čime se efektivno pojednostavljuje model.

Normalizacija ulaza se pokazala nepotrebnom i uzrokovala je eksploziju gradijenata u prvoj epohi jer su naši ulazi dovoljno maleni brojevi (između 0.2 i 0.45) pa ih normalizacija zapravo „širi” u kinematičkoj domeni.

Regularizacija također nije pokazala nikakav utjecaj na sposobnost neuronske mreže. Opaženo je da čak i izrazito male regularizacijske konstante λ dovode do izrazitog pojednostavljivanja modela, a ne rješava se problem iščezavajućih gradijenata zbog čega u sljedećim dijelovima ovog rada ne koristimo regularizaciju. Za dio softvera koji je vezan uz neuronske mreže, Tensorflow omogućava računanje na grafičkoj procesorskoj jedinici (engl. *graphics processing unit*, GPU) što često rezultira znatnim ubrzanjem procesa učenja. Međutim, u našem slučaju nije zapažena nikakva promjena vremena izvršavanja u odnosu na centralnu procesorsku jedinicu (CPU).

Valja ponovno naglasiti i važnost inicijalizacije parametara. Kao i za druge aktivacijske funkcije, postoji preferirani način inicijalizacije kod korištenja SELU aktivacije koji približno čuva varijancu težina kroz slojeve.

4.3 Rezultati prilagodbe neuronskih mreža na opservable

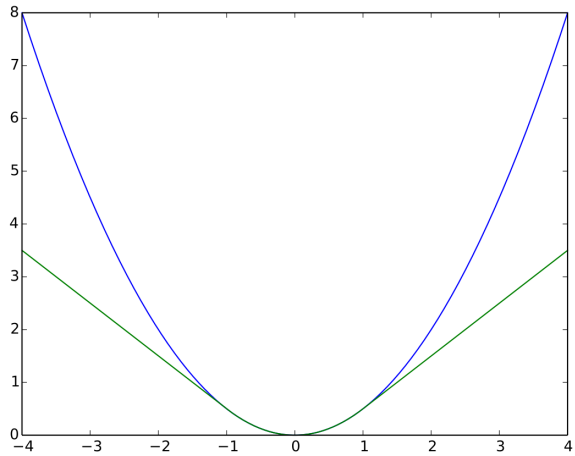
Sada kada je načelno izabrana arhitektura i neke opće značajke mreže, možemo se posvetiti nekim specifičnijim obilježjima i usporediti performanse. Kao što je u nekoliko navrata diskutirano, CFF modeli u [18] su implementirani na način da je svaka CFF funkcija parametrizirana svojom neuronskom mrežom, što daje više parametara, ali i dozu fleksibilnosti da se CFF-ovi uče nezavisno jedni od drugih. Na takav tip modela referirat ćemo se kao *odvojeni* (engl. separated).

S druge strane, među CFF-ovima postoje određene korelacije te je moguće da se odvojenim modelima one izgube pa ih je možda lakše naučiti modelima gdje svaka CFF funkcija predstavlja neuron u izlaznom sloju, odnosno CFF-ovi imaju veliki dio zajedničkih parametara. Na taj tip modela referirat ćemo se kao *dijeljeni* (engl. shared). Potencijalni problem s dijeljenim pristupom je linearna aktivacijska funkcija u izlaznom sloju. To znači da su različiti CFF-ovi ustvari linearne kombinacije jednih te istih funkcija koje generira zadnji skriveni sloj što možebitno smanjuje fleksibilnost modela.

Također, osim prosječne kvadratne pogreške (engl. *mean squared error*, MSE) kao ciljne funkcije za optimizaciju, koristimo i takozvani ,,Huber" gubitak definiran kao:

$$L_\delta(x) = \begin{cases} \frac{1}{2}a^2, & |a| < \delta \\ \delta(|a| - \frac{1}{2}\delta), & \text{inače} \end{cases} \tag{4.3}$$

i koji je prikazan na slici 4.9.



Slika 4.9: Huber gubitak (zeleno) za $\delta = 1$ i kvadratna funkcija gubitka (plavo). Preuzeto iz [25].

Sa slike i iz definicije primjećuje se da je Huber gubitak jednak kvadratnome za područje oko 0 definirano parametrom δ, dok je izvan tog područja linearna funkcija. To dovodi do drugačijeg tretmana netipičnih vrijednosti (engl. *outliera*) kojima se mreža tada manje prilagođava. U ovom radu za parametar δ korištena je vrijednost standardne devijacije razlika između predikcija mreže kada je funkcija gubitka kvadratna i pripadnih izmjerenih opservabli.

Uspoređuju se i performanse modela za tri opisane aktivacijske funkcije: RELU(x), SELU(x) i $\tanh(x)$. Sve navedene kombinacije proučavaju se na mjerenjima s triju eksperimenata: CLAS (180 točaka), Hall A (35 točaka) i HERMES (36 točaka).

U ovom dijelu se za modeliranje opservabli koristi svih 8 CFF-ova u vodećem doprinosu jer je tada primijećeno smanjenje χ^2/N vrijednosti što se vidi iz rezultata svih 36 razmatranih kombinacija koji su dani za *dijeljeni* i *odvojeni* tip modela respektivno u tablicama 4.2 i 4.3.

Funkcija gubitka	MSE			Huber		
Aktivacijska funkcija	SELU	RELU	tanh	SELU	RELU	tanh
CLAS	**0.54**	0.68	**0.54**	0.55	0.61	0.56
Hall A	0.63	**0.54**	0.74	0.66	0.61	0.74
HERMES	0.53	**0.50**	0.52	0.54	**0.50**	0.52

Tablica 4.2: Vrijednosti χ^2/N *dijeljeni* tip modela ovisno o funkciji gubitka, aktivacijskoj funkciji i eksperimentu.

Funkcija gubitka	MSE			Huber		
Aktivacijska funkcija	SELU	RELU	tanh	SELU	RELU	tanh
CLAS	0.80	0.92	1.24	**0.78**	**0.78**	0.95
Hall A	0.71	0.65	0.75	0.71	**0.53**	0.71
HERMES	0.50	**0.47**	0.54	0.51	0.50	0.53

Tablica 4.3: Vrijednosti χ^2/N *odvojeni* tip modela ovisno o funkciji gubitka, aktivacijskoj funkciji i eksperimentu.

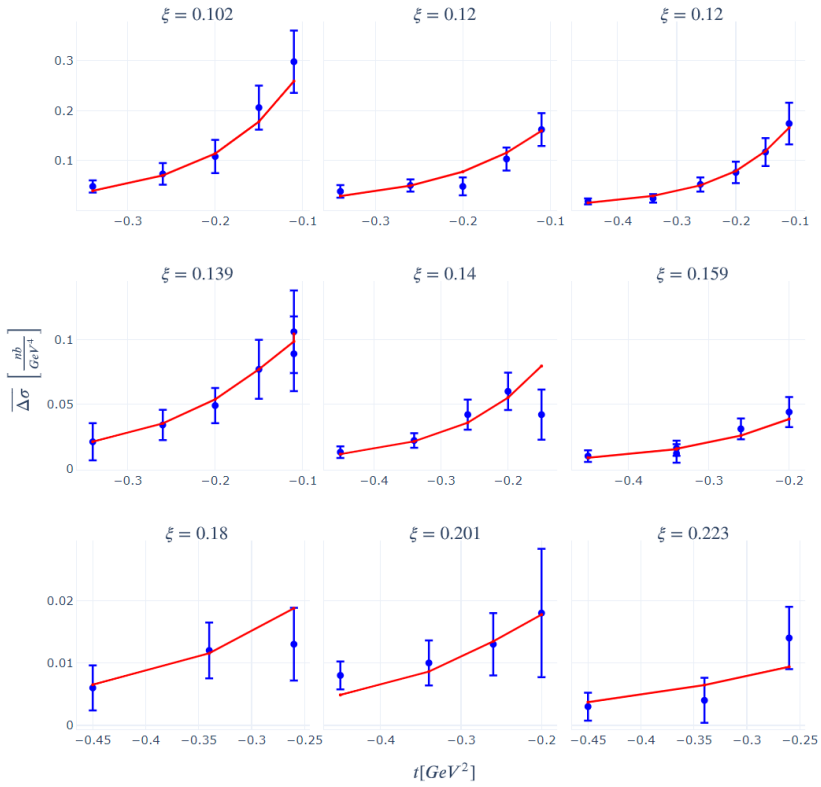
Za CLAS podatke najbolje rezultate daje dijeljeni tip mreže u kombinaciji s kvadratnim gubitkom, za Hall A podatke slične rezultate daju dijeljeni tip s kvadratnim gubitkom i odvojeni tip s Huber gubitkom, dok za HERMES podatke sva 4 podslučaja daju otprilike podjednake rezultate.

U velikoj većini slučajeva, RELU i SELU aktivacije dovode do boljih rezultata u odnosu na tangens hiperbolni. Huber gubitak također ne donosi poboljšanja u dijeljenom tipu, dok za odvojeni tip modela ukupno daje malo bolje rezultate od kvadratnog gubitka.

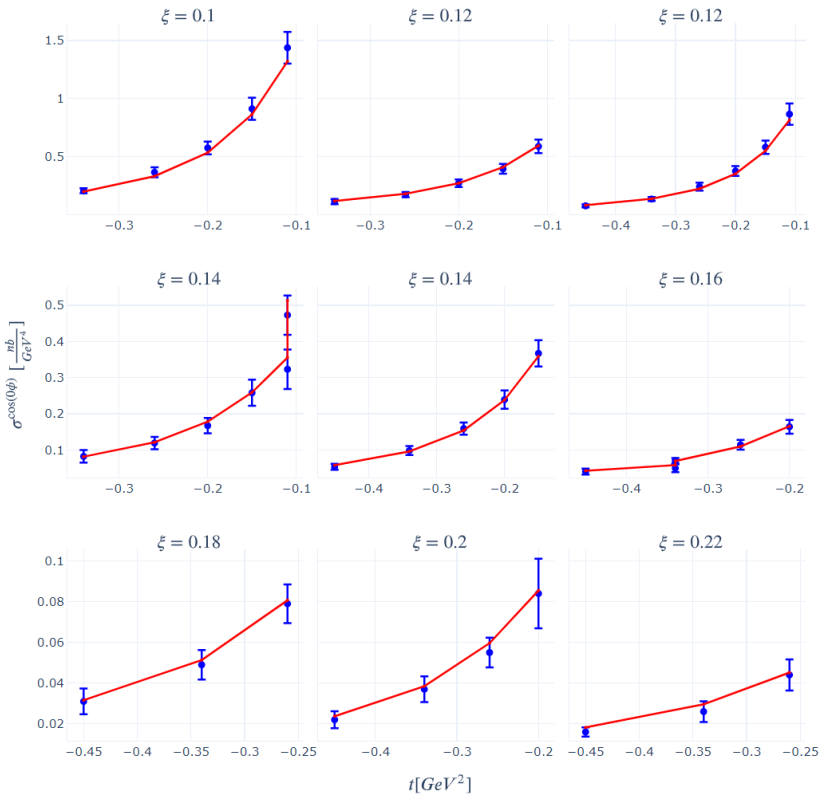
Vidimo da uključenje $Re\tilde{\mathcal{H}}$ i $Re\tilde{\mathcal{E}}$ kao dva dodatna ulaza u modeliranje opservabli zaista dovodi do smanjenja χ^2/N. Male vrijednosti mogu značiti prenaučenost (overfitting) ili da su eksperimentalne ovisnosti precijenjene.

U cilju rasvjetljenja te dileme, na slikama 4.10 i 4.11 prikazana su mjerenja BSDw i BSSw0 opservabli s CLAS eksperimenta te prilagodbu neuronske mreže dijeljenog tipa na iste. Na slikama plavi stupci pogreške označavaju pripadajuće eksperimentalne neodređenosti.

Za podskup koji odgovara prikazanim mjerenjima opservable BSDw $\chi^2/N = 0.36$, dok za podskup koji odgovara prikazanim mjerenjima opservable BSSw0 $\chi^2/N = 0.20$. Uočavamo da je prilagodba na BSSw0 zaista kvalitetnija što odgovara manjoj vrijednosti χ^2/N. Obje vrijednosti su relativno male, stoga možemo zaključiti da se vjerojatno radi o slučaju precjenjivanja eksperimentalnih neodređenosti što smanjuje vrijednost χ^2 statistike. Ovakav prikaz omogućava nam to što je kinematička domena eksperimenta rešetkasta, odnosno postoji više mjerenja za iste vrijednosti ξ te isto tako više mjerenja za iste vrijednosti t. Druge opservable mjerene na CLAS eksperimentu (sve osim BSD i BSS) te sva mjerenja na ostalim eksperimentima (Hall A i HERMES) nemaju to svojstvo.



Slika 4.10: Mjerenja BSDw opservable i njihove eksperimentalne neodređenosti (plavo) i predikcije neuronske mreže (crveno, SELU aktivacija, MSE pogreška).



Slika 4.11: Mjerenja BSSw0 opservable i njihove eksperimentalne neodređenosti (plavo) i predikcije neuronske mreže (crveno, SELU aktivacija, MSE pogreška).

Kako su 3D slike nezgrapne i teže čitljive, prosječne vrijednosti sa standardnim devijacijama za χ^2/N za sve opservable (12 modela za svaku) u svakom eksperimentu dane su u tablici 4.4.

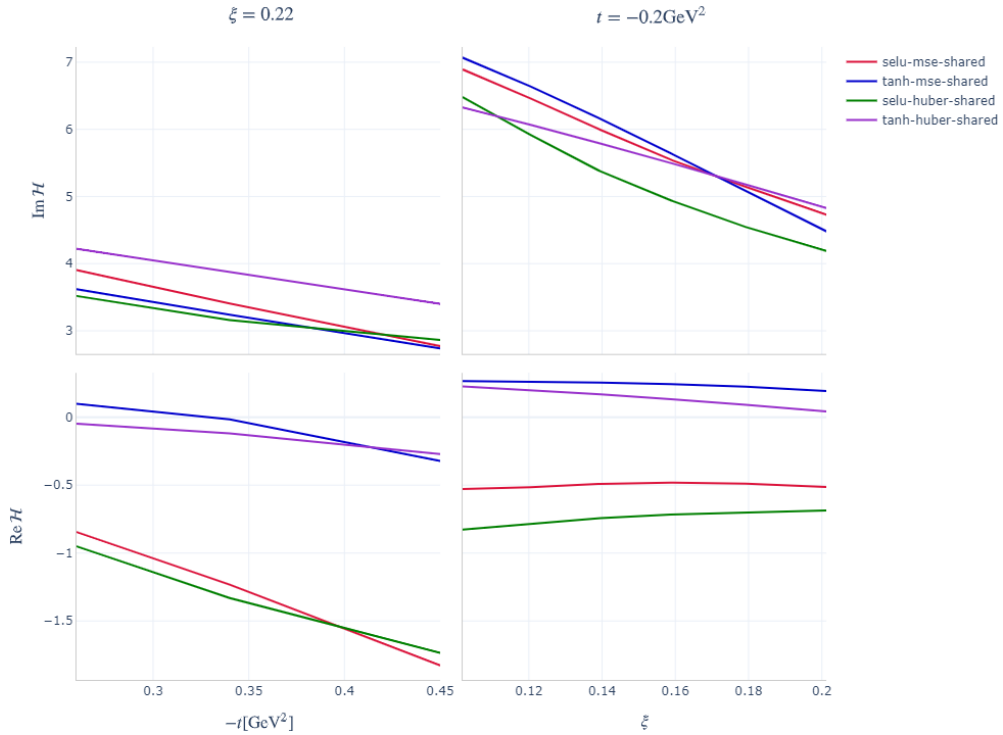
opservabla	CLAS	Hall A	HERMES
BSDw	0.43$\pm$0.04	0.52$\pm$0.02	
BSSw0	0.41$\pm$0.30	0.67$\pm$0.15	
BSSw1	1.10$\pm$0.40	0.88$\pm$0.20	
BSA	2.31$\pm$0.28		
BTSAw0	0.93$\pm$0.15		
BTSAw1	0.72$\pm$0.33		
TSA	**0.33$\pm$ 0.06**		
BCA0			0.80$\pm$0.05
BCA1			**0.23$\pm$ 0.06**

Tablica 4.4: Prosječne vrijednosti χ^2/N sa standardnim devijacijama za sve korištene opservable.

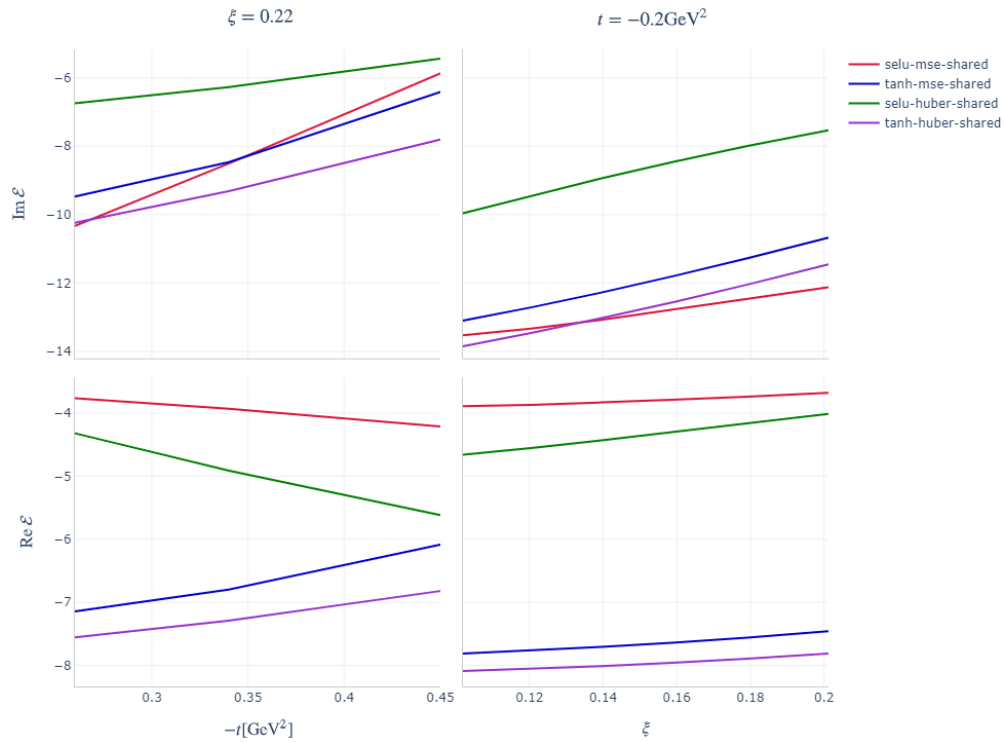
Valja primijetiti da se najmanje vrijednosti dobivaju za opservable TSA i BCA1 što se indirektno poklapa s rezultatima iz [18] i [19] da se CFF funkcije najbolje ekstrahiraju kada se koristi kombinacija tih dvaju opservabli. Naime, niži χ^2 znači bolju prilagodbu na podatke što implicira da su CFF funkcije koje su dobivene kao „nusprodukt" procesa prilagodbe bliže stvarnim CFF funkcijama koje generira „priroda" nego one s višim χ^2.

4.4 Ekstrahirane CFF funkcije

U ovom potpoglavlju dani su najvažniji rezultati ovog istraživanja. Za svaki od 3 eksperimenta gdje su mjerene korištene opservable, donosimo sve 4 CFF funkcije u vodećem doprinosu. CFF funkcije prikazane su u panelima, svaki od kojih ima po 4 dijela: imaginarni i realni dio za konstantnu vrijednost ξ (x_B) i t. Na svakom od ta 4 dijela dana su po 4 izabrana modela koja su označena na legendi u formatu "aktivacija-pogreška-tip_modela". Na taj način vidimo kako odabir određenih značajki utječe na oblik ekstrahirane funkcije. Za CLAS mjerenja, na slikama 4.12 do 4.15 prikazane se funkcije dobijene za modele s najmanjim χ^2 vrijednostima.

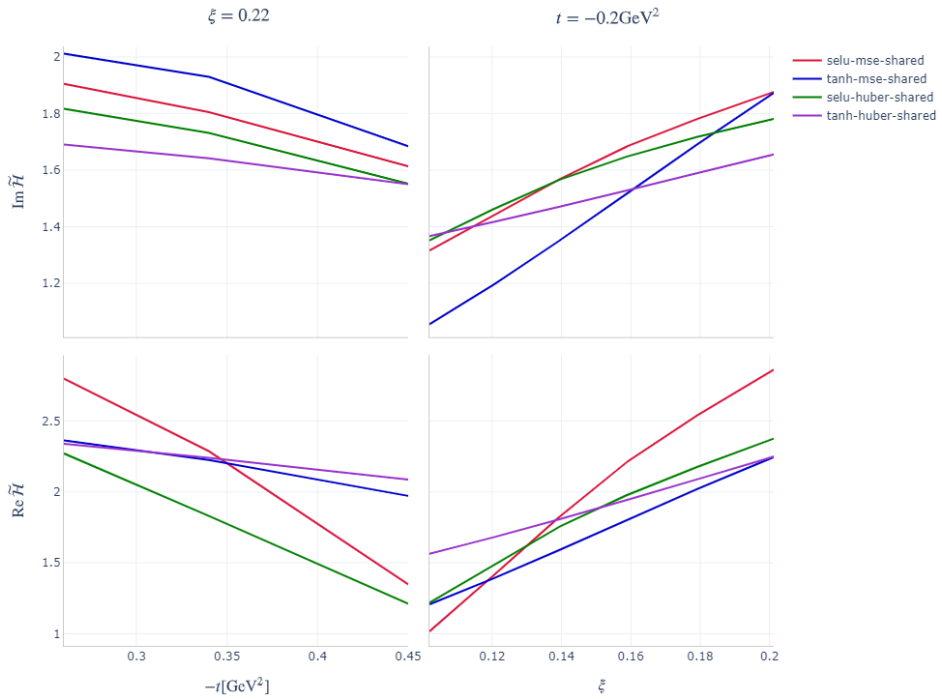


Slika 4.12: $Im\mathcal{H}$ i $Re\mathcal{H}$ ekstrahirane s različitim modelima iz CLAS mjerenja.

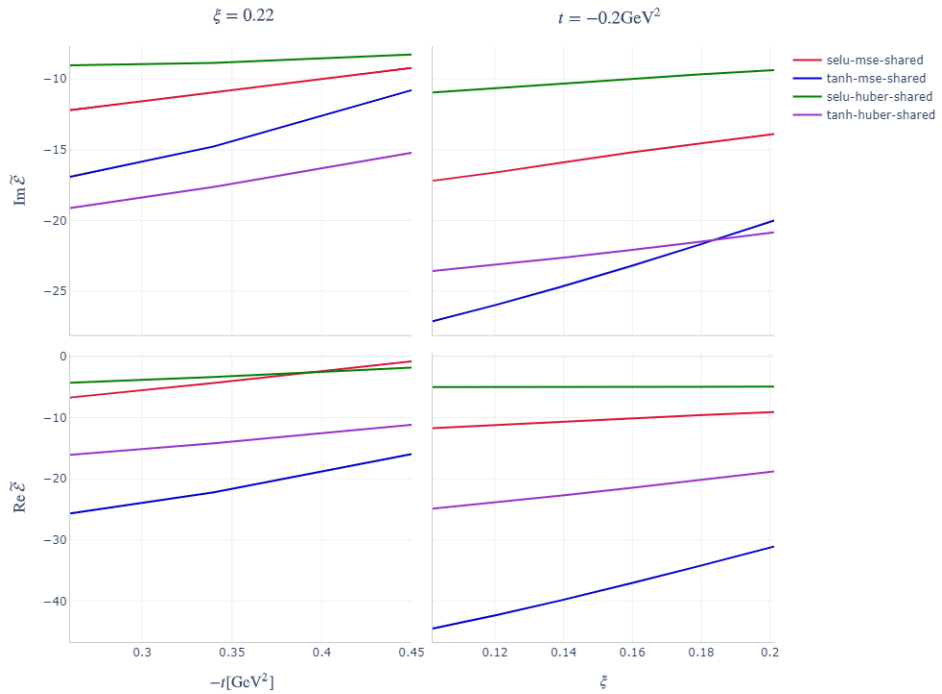


Slika 4.13: $Im\mathcal{E}$ i $Re\mathcal{E}$ ekstrahirane s različitim modelima iz CLAS mjerenja.

Sva četiri modela relativno slično predviđaju oblik $Im\mathcal{E}$ i posebno $Im\mathcal{H}$, koja je općenito najdominantniji i najdostupniji doprinos opservablama. S druge strane, kod $Re\mathcal{H}$ i $Re\mathcal{E}$ vidi segmentacija prema aktivacijskoj funkciji na po dvije slične predikcije.

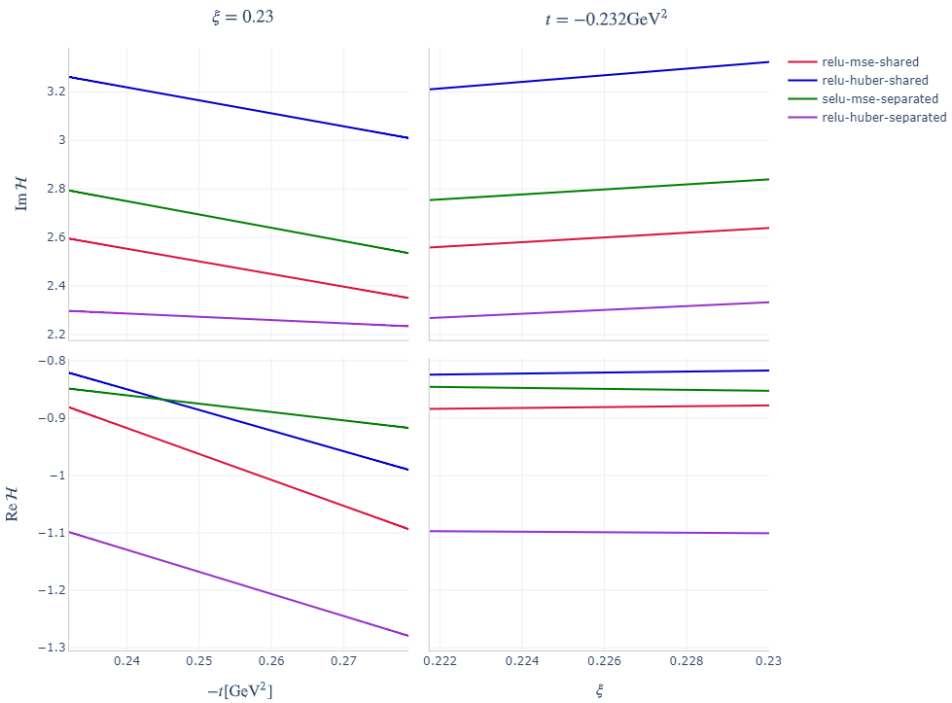


Slika 4.14: $Im\tilde{\mathcal{H}}$ i $Re\tilde{\mathcal{H}}$ ekstrahirane s različitim modelima iz CLAS mjerenja.

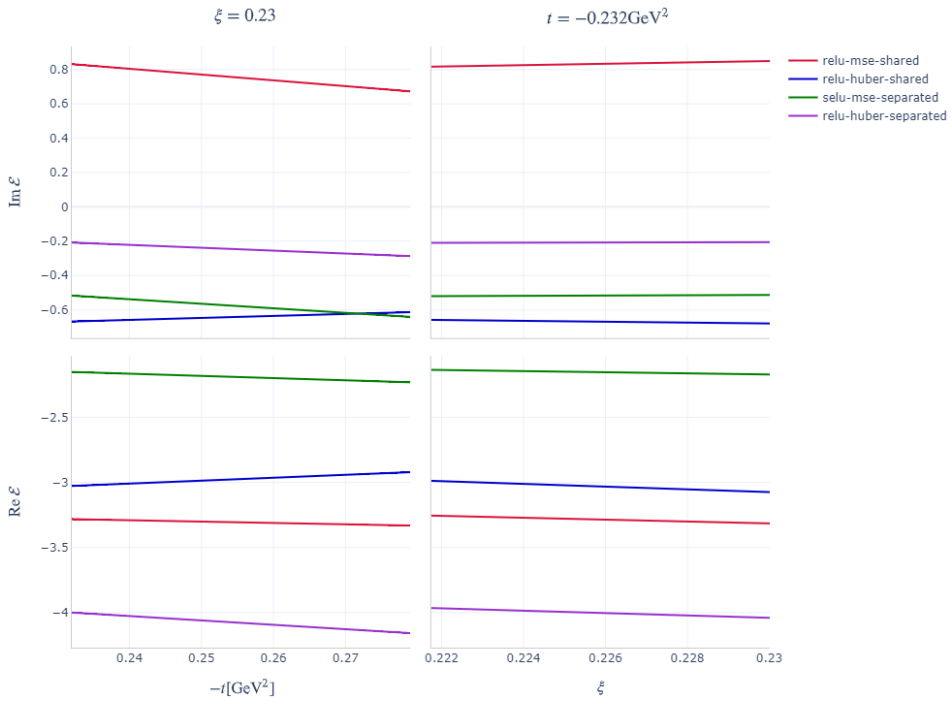


Slika 4.15: $Im\tilde{\mathcal{E}}$ i $Re\tilde{\mathcal{E}}$ ekstrahirane s različitim modelima iz CLAS mjerenja.

Ovdje situacija nalikuje prethodnoj, $Re\tilde{\mathcal{H}}$, a posebno $Im\tilde{\mathcal{H}}$ su predviđene relativno jednoliko sa svim modelima, dok kod $Im\tilde{\mathcal{E}}$ i $Re\tilde{\mathcal{E}}$ ponovno dobivamo blago raslojavanje predikcija većinom po ključu da su iste aktivacijske funkcije bliže. Analogni grafovi ekstrahirani iz Hall A mjerenja, prikazani su na slikama 4.16 do 4.19.

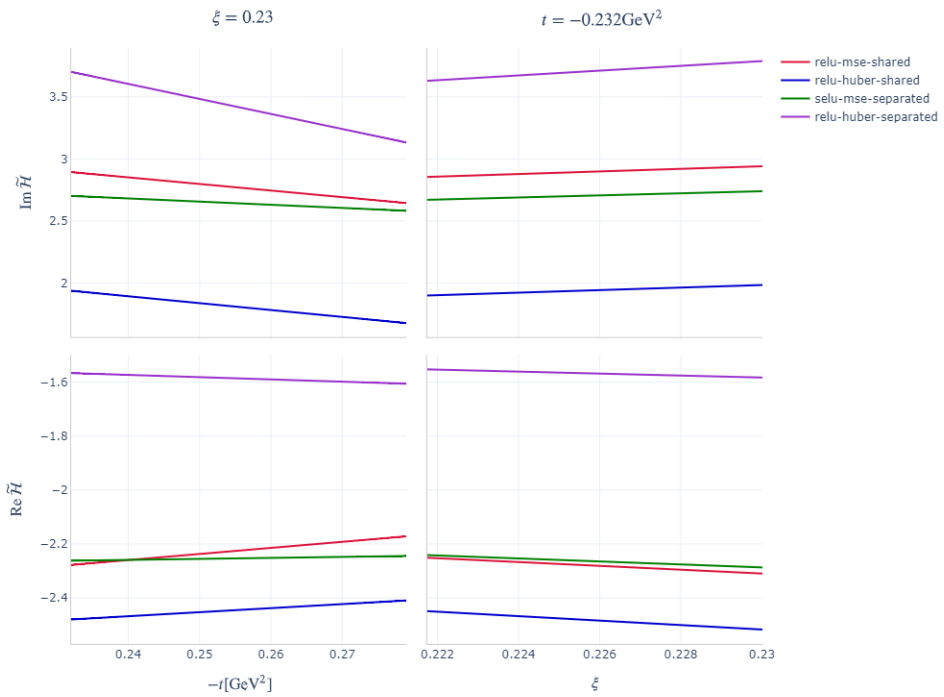


Slika 4.16: $Im\mathcal{H}$ i $Re\mathcal{H}$ ekstrahirane s različitim modelima iz Hall A mjerenja.

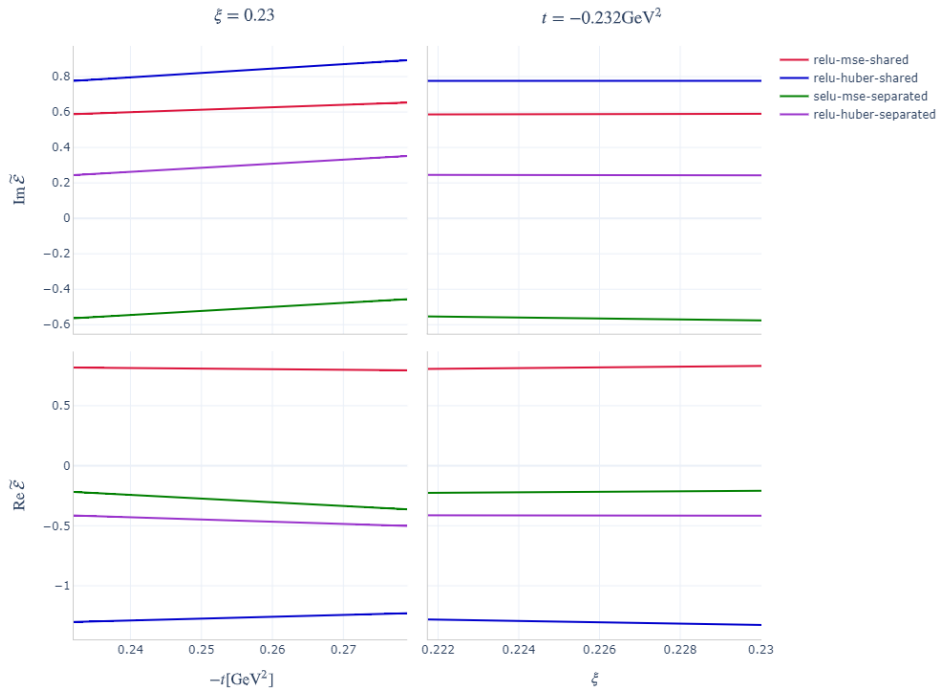


Slika 4.17: $Im\mathcal{E}$ i $Re\mathcal{E}$ ekstrahirane s različitim modelima iz Hall A mjerenja.

Svi modeli predviđaju $Im\mathcal{H}$, $Re\mathcal{H}$ te donekle i $Im\mathcal{E}$ na bliskim vrijednostima, dok se kod $Re\mathcal{E}$ primjećuje malo veće, ali i dalje ne snažno raslojavanje predikcija bez očitog uzorka po modelima. Vrijednosti se, izuzevši $Im\mathcal{E}$, ugrubo poklapaju vrijednostima ekstrahiranim iz CLAS mjerenja na približnoj kinematičkoj domeni.

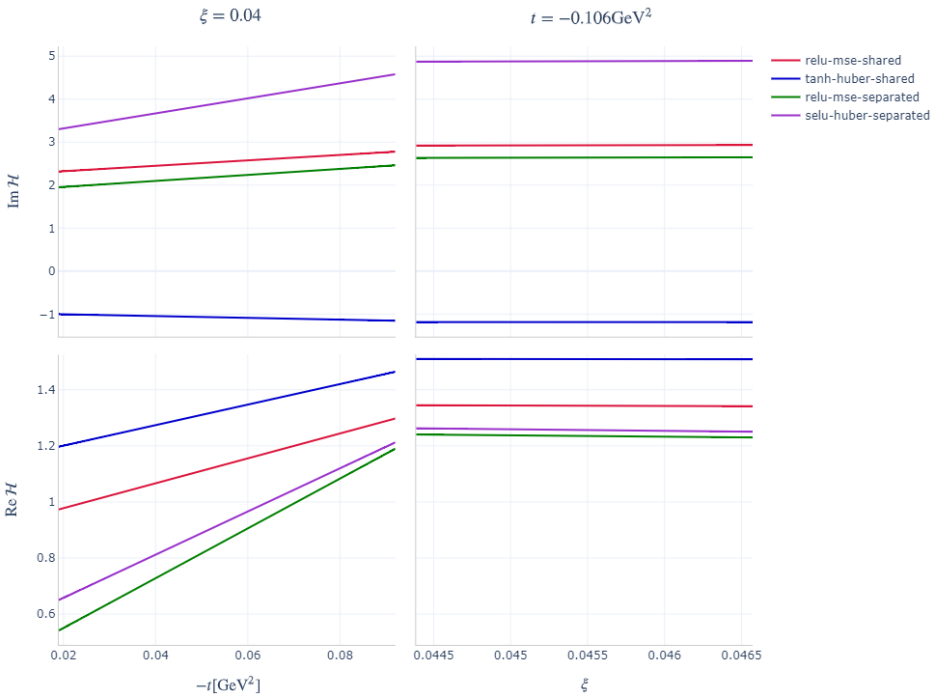


Slika 4.18: $Im\tilde{\mathcal{H}}$ i $Re\tilde{\mathcal{H}}$ ekstrahirane s različitim modelima iz HAll A mjerenja.

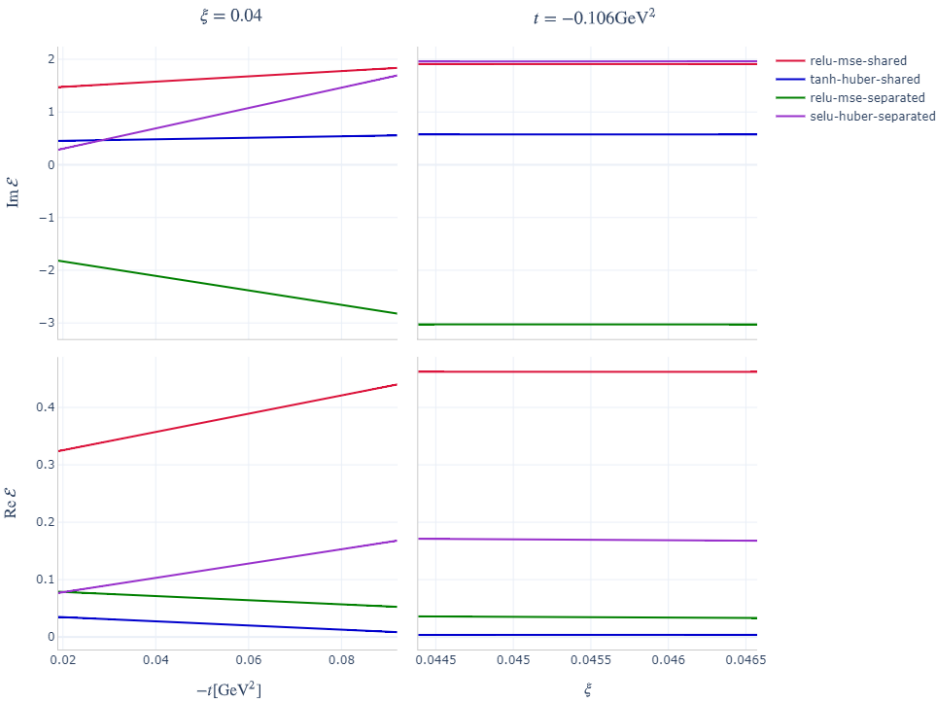


Slika 4.19: $Im\tilde{\mathcal{E}}$ i $Re\tilde{\mathcal{E}}$ ekstrahirane s različitim modelima iz HAll A mjerenja.

U ovom slučaju, osim $Im\tilde{\mathcal{H}}$, vrijednosti CFF funkcija ne poklapaju se s onima ekstrahiranima iz CLAS mjerenja, dok $Im\tilde{\mathcal{E}}$ i $Re\tilde{\mathcal{E}}$ izgledaju usko raspršeni oko nule bez uzorka po modelima. Konačno, analogni prikazi CFF funkcija ekstrahiranih iz HERMES mjerenja dani su na slikama 4.20 do 4.23.

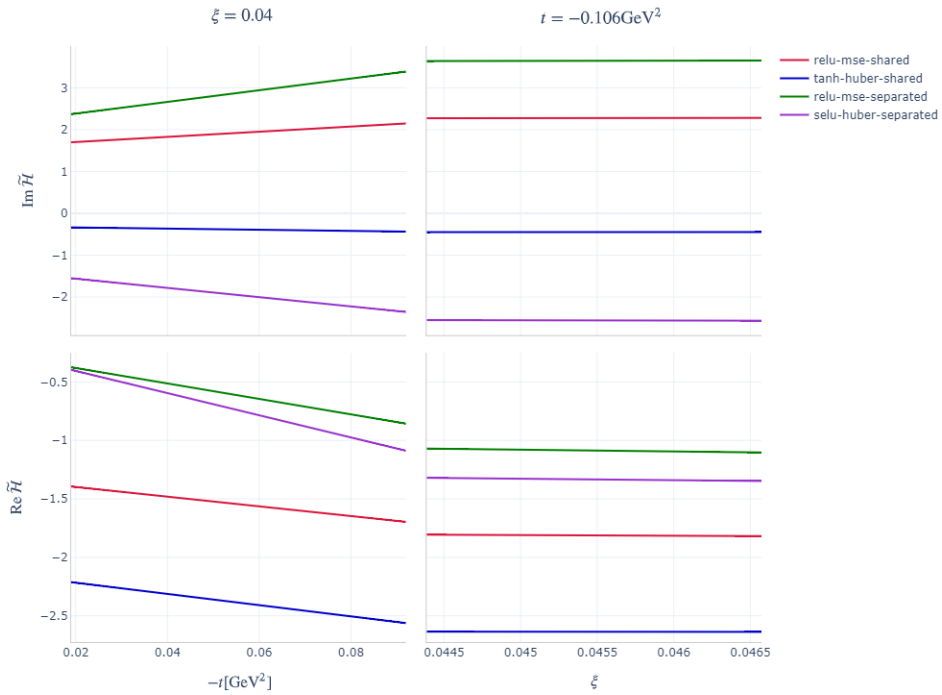


Slika 4.20: $Im\mathcal{H}$ i $Re\mathcal{H}$ ekstrahirane s različitim modelima iz HERMES mjerenja.

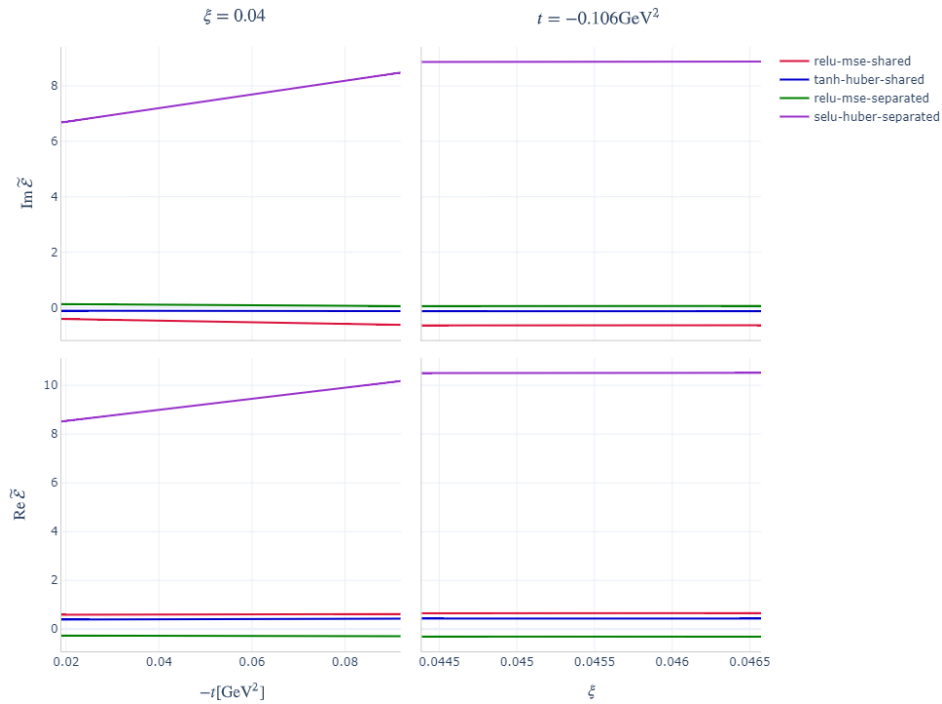


Slika 4.21: $Im\mathcal{E}$ i $Re\mathcal{E}$ ekstrahirane s različitim modelima iz HERMES mjerenja.

U ovom slučaju, nemamo usporedivu kinematičku domenu s ostala dva seta mjerenja, zbog čega nisu moguće usporedbe vrijednosti ekstrahiranih CFF funkcija. Predikcije za $Im\mathcal{H}$ i $Im\mathcal{E}$ su raspršene na nešto šire raspone vrijednosti u odnosu na $Re\mathcal{H}$ i $Re\mathcal{E}$.



Slika 4.22: $Im\tilde{\mathcal{H}}$ i $Re\tilde{\mathcal{H}}$ ekstrahirane s različitim modelima iz HERMES mjerenja.



Slika 4.23: $Im\tilde{\mathcal{E}}$ i $Re\tilde{\mathcal{E}}$ ekstrahirane s različitim modelima iz HERMES mjerenja.

Kod $Im\tilde{\mathcal{H}}$ i $Re\tilde{\mathcal{H}}$ vidimo da su predikcije blago raspršene, dok kod predikcija za $Im\tilde{\mathcal{E}}$ i $Re\tilde{\mathcal{E}}$ jedan model odskače od ostalih koji su usko oko nule.

5 Zaključak

U ovom radu proučene su različiti tipovi i arhitekture neuronskih mreža u cilju što bolje prilagodbe na DVCS mjerenja s različitih eksperimenata. Opisane su GPD funkcije kao objekti koji sadrže trodimenzionalnu informaciju o strukturi nukleona, distribuciju naboja u transverzalnoj ravnini, distribuciju longitudinalnog impulsa i njihove korelacije. Opisan je DVCS proces kao teorijski najčišći pristup GPD-ovima, kao i integrali GPD-ova, komptonski form faktori o kojima direktno ovisi DVCS amplituda.

Opisane su i neuronske mreže, skup algoritama kojima modeliramo CFF-ove na način da ne unose pristranost. Dana je njihova biološka motivacija te povijesni razvoj od perceptrona do danas. Također je preko algoritma propagacije unatrag opisan način njihovog učenja i optimizacije parametara. Diskutirane su i standardne poteškoće koje se javljaju pri učenju tzv. dubokih mreža.

Na koncu se optimalno izabrana arhitektura neuronske mreže primijenila na stvarna mjerenja DVCS opservabli s tri različita eksperimenta. Kvaliteta prilagodbe ocjenjivala se standardnim χ^2 testom, ali i vizualno. Tijekom tog postupka, ekstrahirano je svih 8 CFF funkcija u vodećem doprinosu te su uspoređeni funkcijski oblici dobiveni različitim izborima aktivacijske funkcije, funkcije pogreške i tipa neuronske mreže.

Dodaci

Ovdje donosimo *Python* skripte prilagodbe „GeParD" softvera na moderniju biblioteku *Tensorflow*. Kostur koda i struktura izvršavanja zadaća posuđeni su iz [18], dok su u sklopu ovog rada implementirane nove funkcionalnosti na osnovu istog.

U dodatku A prikazana je skripta *load_data.py* koja služi za učitavanje željenih setova podataka. Dodatak B sadrži dvije varijante skripte *models.py* u kojoj se kreira graf operacija za dva opisana tipa neuronske mreže: dijeljeni i odvojeni. U dodatku C nalazi se skripta *train.py* koja šalje podatke kroz stvoreni graf operacija, evaluira pogrešku i na kraju svake epohe izvršava validaciju. Na koncu, u dodatku D dana je skripta *visualize_CFFs.py* koja služi za prikaz ekstrahiranih CFF funkcija. Osim koda danog u nastavku, korišten je i alat *checkmate* [26] koji pamti instancu modela s najnižom pogreškom na skupu za validaciju.

Dodatak A load_data.py

Kod 1: load_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, 'nn-shared')
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 *

def get_dataset(dataset, seed, input_norm=False):
    """
    Return real data to be fitted to
    Args:
        dataset (string): defines which dataset or combination
            of dataset is used for further training;
            see down for actual names
        random_state (int): ensures reproducibility of results by
            splitting data the same way
        input_norm (Boolean): if True, input coordinates (t and /xi)
            are returned normalized - with mean=0 and var=1
    Returns:
        pts  (DataSet)  : all data used
        fitpoints_train (list): List of datapoints used
            for training.
        fitpoints_val (list): List of datapoints used
```

```
            for validation.
    """
    if dataset == 'BSA-CLAS':
        pts = CLAS14BSApts
    elif dataset == 'TSA-CLAS':
        pts = CLAS14TSApts
    elif dataset == 'BTSA-CLAS':
        pts = CLAS14BTSApts
    elif dataset == 'BSDw-CLAS':
        pts = C_BSDwpts
    elif dataset == 'BSSw0-CLAS':
        pts = C_BSSw0pts
    elif dataset == 'BSSw1-CLAS':
        pts = C_BSSw1pts
    elif dataset == 'BSDw-HALL_A':
        pts = H_BSDwpts
    elif dataset == 'BSSw0-HALL_A':
        pts = H_BSSw0pts
    elif dataset == 'BSSw1-HALL_A':
        pts = H_BSSw1pts
    elif dataset == 'BCA0-HERMES':
        pts = BCA0points
    elif dataset == 'BCA1-HERMES':
        pts = BCA1points
    elif dataset == 'CLAS':
        pts = CLAS14TSApts + CLAS14BTSApts + C_BSSw1pts + C_BSSw0pts \
        + C_BSDwpts + CLAS14BSApts # CLAS 180 točaka
    elif dataset == 'HALL_A':
        pts = H_BSDwpts + H_BSSw1pts + H_BSSw0pts # HALL_A 35 točaka
    elif dataset == 'BCA-all':
        pts = BCA0points + BCA1points
    elif dataset == 'CLAS + HALL_A':
        pts = CLAS14TSApts + CLAS14BTSApts + C_BSSw1pts + C_BSSw0pts  \
        + C_BSDwpts + CLAS14BSApts + H_BSDwpts + H_BSSw1pts + H_BSSw0pts

    fit = []
    for pt in pts:
        if pt.y1name in ['BSDw', 'BSSw', 'BSA']:
            pt.in1polarizationvector = 'L'
            pt.in1polarization = 1
            pt.in2polarization = 0
        if pt.y1name == 'TSA':
            pt.in1polarization = 0
            pt.in2polarizationvector = 'L'
            pt.in2polarization = 1
        if pt.y1name == 'BTSA':
            pt.in1polarizationvector = 'L'
            pt.in1polarization = 1
            pt.in2polarizationvector = 'L'
            pt.in2polarization = 1
        if pt.y1name == 'BCA':
            pt.in1polarization = 0
            pt.in2polarization = 0
            pt.in1charge = -1
        fit.append(pt)

    fitpoints_train, fitpoints_val = train_test_split(fit, test_size=0.25,
                                                      random_state=seed)

    if input_norm:

        ts=[x.t for x in fitpoints_train]
        xis=[x.xi for x in fitpoints_train]

        mt=np.array(ts).mean()
        mxi=np.array(xis).mean()
```

```
        vt=np.array(ts).var()
        vxi=np.array(xis).var()

        for pt in fitpoints_train:
            pt.t=(pt.t-mt)/np.sqrt((vt+epsilon))
            pt.xi=(pt.xi-mxi)/np.sqrt((vxi+epsilon))

        for pt in fitpoints_val:
            pt.t=(pt.t-mt)/np.sqrt((vt+epsilon))
            pt.xi=(pt.xi-mxi)/np.sqrt((vxi+epsilon))

    return pts, fitpoints_train, fitpoints_val
```

Dodatak B models.py

Kod 2: nn-separated/models.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

# 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, 'nn-separated')
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 ToyModel(Model_new.ComptonFormFactors, Model_new.ElasticDipole):
    """
    A model for all 8 leading order CFF functions.
    """

    def __init__(self, placeholder,  CFFs=[], num_neurons_per_layers=[50,50],
                 initializer=(tf.keras.initializers
                              .VarianceScaling(scale=1.0, mode='fan_in')),
                 activation=tf.nn.selu):
        """
        Initialize the model and store placeholders for later usage.
        Args:
            placeholder (tf.placeholder): A placeholder through which the
                CFF functions acquire inputs.
            CFFs (list): List of CFFs user wants to include in modeling
                the cross sections.
            num_neurons_per_layers (list): List of numbers of neurons per
                each hidden layer.
            initializer (tf.initializer): Initializer for weights and biases.
            activation: activation function used in every layer of NN.
        """
        Model_new.ComptonFormFactors.__init__(self)
        Model_new.ElasticDipole.__init__(self)
```

```
        self.input = placeholder
        assert len(num_neurons_per_layers) > 0
        self.num_neurons_per_layers = num_neurons_per_layers
        self.initializer = initializer
        self.activation = activation
        self.CFFs = CFFs

        self._ImH_create()
        self._ReH_create()
        self._ImE_create()
        self._ReE_create()
        self._ImHt_create()
        self._ReHt_create()
        self._ImEt_create()
        self._ReEt_create()

    def _ImH_create(self):
        """
        Create neural network for ImH CFF function.
        """
        ImH_model = tf.keras.Sequential(name='ImH_model')

        for num_neurons in self.num_neurons_per_layers:
            ImH_model.add(tf.keras.layers.Dense(num_neurons, activation=self.activation,
                                                kernel_initializer=self.initializer))
        ImH_model.add(tf.keras.layers.Dense(1, kernel_initializer=self.initializer))
        self._ImH = ImH_model(self.input)

    def ImH(self, pt):
        """
        Calculate ImH.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ImH' in self.CFFs:
            return self._ImH
        else:
            return 0.

    def _ReH_create(self):
        """
        Create neural network for ReH CFF function.
        """
        ReH_model = tf.keras.Sequential(name='ReH_model')

        for num_neurons in self.num_neurons_per_layers:
            ReH_model.add(tf.keras.layers.Dense(num_neurons, activation=self.activation,
                                                kernel_initializer=self.initializer))
        ReH_model.add(tf.keras.layers.Dense(1, kernel_initializer=self.initializer))
        self._ReH = ReH_model(self.input)

    def ReH(self, pt):
        """
        Calculate ReH.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ReH' in self.CFFs:
            return self._ReH
        else:
            return 0.

    def _ImE_create(self):
        """
        Create neural network for ImE CFF function.
        """
```

```
        ImE_model = tf.keras.Sequential(name='ImE_model')

        for num_neurons in self.num_neurons_per_layers:
            ImE_model.add(tf.keras.layers.Dense(num_neurons, activation=self.activation,
                                                kernel_initializer=self.initializer))
        ImE_model.add(tf.keras.layers.Dense(1, kernel_initializer=self.initializer))
        self._ImE = ImE_model(self.input)

    def ImE(self, pt):
        """
        Calculate ImE.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ImE' in self.CFFs:
            return self._ImE
        else:
            return 0.

    def _ReE_create(self):
        """
        Create neural network for ReE CFF function.
        """
        ReE_model = tf.keras.Sequential(name='ReE_model')

        for num_neurons in self.num_neurons_per_layers:
            ReE_model.add(tf.keras.layers.Dense(num_neurons, activation=self.activation,
                                                kernel_initializer=self.initializer))
        ReE_model.add(tf.keras.layers.Dense(1, kernel_initializer=self.initializer))
        self._ReE = ReE_model(self.input)

    def ReE(self, pt):
        """
        Calculate ReE.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ReE' in self.CFFs:
            return self._ReE
        else:
            return 0.

    def _ImHt_create(self):
        """
        Create neural network for ImHt CFF function.
        """
        ImHt_model = tf.keras.Sequential(name='ImHt_model')

        for num_neurons in self.num_neurons_per_layers:
            ImHt_model.add(tf.keras.layers.Dense(num_neurons, activation=self.activation,
                                                 kernel_initializer=self.initializer))
        ImHt_model.add(tf.keras.layers.Dense(1, kernel_initializer=self.initializer))
        self._ImHt = ImHt_model(self.input)

    def ImHt(self, pt):
        """
        Calculate ImHt.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ImHt' in self.CFFs:
            return self._ImHt
        else:
            return 0.

    def _ReHt_create(self):
```

```
        """
        Create neural network for ReHt CFF function.
        """
        ReHt_model = tf.keras.Sequential(name='ReHt_model')

        for num_neurons in self.num_neurons_per_layers:
            ReHt_model.add(tf.keras.layers.Dense(num_neurons, activation=self.activation,
                                                 kernel_initializer=self.initializer))
        ReHt_model.add(tf.keras.layers.Dense(1, kernel_initializer=self.initializer))
        self._ReHt = ReHt_model(self.input)

    def ReHt(self, pt):
        """
        Calculate ReHt.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ReHt' in self.CFFs:
            return self._ReHt
        else:
            return 0.

    def _ImEt_create(self):
        """
        Create neural network for ImEt CFF function.
        """
        ImEt_model = tf.keras.Sequential(name='ImEt_model')

        for num_neurons in self.num_neurons_per_layers:
            ImEt_model.add(tf.keras.layers.Dense(num_neurons, activation=self.activation,
                                                 kernel_initializer=self.initializer))
        ImEt_model.add(tf.keras.layers.Dense(1, kernel_initializer=self.initializer))
        self._ImEt = ImEt_model(self.input)

    def ImEt(self, pt):
        """
        Calculate ImEt.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ImEt' in self.CFFs:
            return self._ImEt
        else:
            return 0.

    def _ReEt_create(self):
        """
        Create neural network for ReEt CFF function.
        """
        ReEt_model = tf.keras.Sequential(name='ReEt_model')

        for num_neurons in self.num_neurons_per_layers:
            ReEt_model.add(tf.keras.layers.Dense(num_neurons, activation=self.activation,
                                                 kernel_initializer=self.initializer))
        ReEt_model.add(tf.keras.layers.Dense(1, kernel_initializer=self.initializer))
        self._ReEt = ReEt_model(self.input)

    def ReEt(self, pt):
        """
        Calculate ReEt.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ReEt' in self.CFFs:
            return self._ReEt
        else:
```

```
            return 0.

class Regularizer(tf.keras.regularizers.Regularizer):
    def __init__(self, strength):
        self.strength = strength

    def __call__(self, x):
        return self.strength * tf.reduce_sum(tf.square(x))

def create_graph(point_example, lr, num_neurons_per_layers, loss='mse',
                 activation='Selu',
                 reg_constant=0.0,
                 CFFs=['ImH', 'ReH', 'ReHt', 'ImHt', 'ImE', 'ReE', 'ReEt', 'ImEt']):
    """
    Create graph for learning the CFF functions.
    Args:
        point_example (DataPoint): One datapoint from some dataset for
            the placeholder to copy.
        lr (float): Learning rate.
        num_neurons_per_layers (list): List of numbers of neurons per each hidden layer.
        loss (string): 'mse' or 'huber' - defines loss function.
        activation (string): 'Selu', 'Relu' or 'Tanh'
        reg_constant (float): Regularization constant.
        CFFs (list): List of CFFs user wants to include in modeling
            the cross sections.

    Returns:
        point (tf.placeholder): Point placeholder.
        y_true (tf.placeholder): Placeholder for the true value of
            observable for a given point.
        partial_loss (tf.Tensor): Partial loss function tensor.
        train_ops (dict): A dictionary containing training operations for
            different combinations of CFF functions.
        predictions (dict): Dictionary of predictions for each CFF function.
        global_step_tensor (tf.Tensor): Global step tensor.
        merged (tf.Tensor):  Tensor of type string resulting from the merging
            of summaries tracked for visualizing in Tensorboard.
        y_hat (tf.Tensor): Tensor for observable prediction.
        chi (tf.Tensor): Tensor used for calculating \chi^2 statistic.
    """
    tf.reset_default_graph()
    with tf.device('/CPU:0'):

        with tf.name_scope('point'):
            point = copy.copy(point_example) # can be any point, I think
            # Placeholders for inputs
            point.t = tf.placeholder(tf.float32, name='t')
            point.xB = tf.placeholder(tf.float32, name='xB')
            point.y1name = tf.placeholder(tf.string, name='y1name')
            point.FTn = tf.placeholder(tf.float32, name='FTn')
            point.in1polarization = tf.placeholder(tf.float32, name='in1polarization')
            point.in2polarization = tf.placeholder(tf.float32, name='in2polarization')
            point.in2polarizationvector = tf.placeholder(tf.string, name='in2pvector')
            point.W = tf.placeholder(tf.float32, name='W')
            point.Q2 = tf.placeholder(tf.float32, name='Q2')
            point.eps = tf.placeholder(tf.float32, name='eps')
            point.yaxis = tf.placeholder(tf.string, name='yaxis')
            point.err = tf.placeholder(tf.float32, name='err')
            point.xi = tf.placeholder(tf.float32, name='xi')

        global_step_tensor = tf.train.get_or_create_global_step()
        regularizer=Regularizer(strength=reg_constant)

        with tf.name_scope('Build-model-and-theory'):
            # Create a model given the input data.
            # The same point goes to the model to calculate the CFF functions
```

```
            # and to the th.predict(point). This couldn't be solved
            # differently because the code that calculate observables
            # creates new instance of a Point and replaces my placeholders.
            # So CFF functions all have the same input ([point.t, point.xi])
            # and do not consider pt argument that is passed to them by the
            # Approach.
            if activation == 'Selu':
                act=tf.nn.selu
                init=tf.keras.initializers.VarianceScaling(scale=1.0, mode='fan_in')
            if activation == 'Relu':
                act=tf.nn.relu
                init=tf.keras.initializers.he_normal()
            if activation=='Tanh':
                act=tf.nn.tanh
                init=tf.keras.initializers.glorot_normal()

            m = ToyModel(tf.reshape(tf.stack([point.t, point.xi],0),(1,2)),
                         num_neurons_per_layers=num_neurons_per_layers,
                         CFFs=CFFs, activation=act, initializer=init)
            th = Approach_new.BM10tw2(m)
            th.prepare(point)

        predictions = {'ImH': th.m.ImH(point),
                       'ReH': th.m.ReH(point),
                       'ImE': th.m.ImE(point),
                       'ReE': th.m.ReE(point),
                       'ImHt': th.m.ImHt(point),
                       'ReHt': th.m.ReHt(point),
                       'ImEt': th.m.ImEt(point),
                       'ReEt': th.m.ReEt(point)}

        with tf.name_scope('Loss-definition'):
            y_true = tf.placeholder(tf.float32, name='y_true')
            with tf.name_scope('y_hat'):
                y_hat = th.predict(point)

            # Definition of loss function
            if loss=='mse':
                l=tf.square(y_hat - y_true, name='squared_loss')
            if loss=='huber':
                l=tf.losses.huber_loss(y_true, y_hat, delta=0.025)

            #Logging weights of every defined layer and adding regularization loss term
            reg_loss=0
            for var in tf.global_variables():
                if 'kernel:0' in var.name:
                    tf.summary.histogram("{}/weights"\
                                         .format(var.name.replace(':', '_')), var)
                    reg_loss+=regularizer(var)

            # Definition of a chi^2 value for a point passing through the graph
            chi = tf.math.divide(tf.square(y_hat - y_true),
                                 tf.square(point.err), name='chi')
            #Partial loss
            partial_loss= l + reg_loss

        with tf.name_scope('Train'):
            decayed_lr = tf.train.exponential_decay(lr,
                                        global_step_tensor, 180,
                                        0.97, staircase=True)
            tf.summary.scalar('Decayed_learning_rate', decayed_lr)

            with tf.name_scope('Weights'):
            #Logging weights of every defined layer
                for var in tf.global_variables():
                    if 'kernel:0' in var.name:
```

```
                        tf.summary.histogram("{}/weights"\
                                                .format(var.name.replace(':', '_')), var)

            with tf.name_scope('Activations'):
            #Logging layers' activations
                input_op=(tf.get_default_graph()
                          .get_operation_by_name("Build-model-and-theory/Reshape"))
                tf.summary.histogram("inputs", input_op.outputs[0])
                for i,cff in enumerate(CFFs):
                    for j in range(len(num_neurons_per_layers)):
                        if j==0:
                            op=(tf.get_default_graph()
                                .get_operation_by_name(
                                    "Build-model-and-theory/{}_model/{}"
                                                        .format(cff, activation)))
                            tf.summary.histogram("Activation_{}{}"\
                                                    .format(i+1,j+1), op.outputs[0])
                        else:
                            op=(tf.get_default_graph()
                                .get_operation_by_name(
                                    "Build-model-and-theory/{}_model/{}_{}"
                                                        .format(cff,activation, j)))
                            tf.summary.histogram("Activation_{}{}"\
                                                    .format(i+1,j+1), op.outputs[0])

            grads_and_vars=(tf.train.AdamOptimizer(learning_rate=decayed_lr)
                            .compute_gradients(partial_loss))

            #Logging gradients of every defined layer
            for g, v in grads_and_vars:
                if g is not None and 'bias' not in v.name:
                    tf.summary.histogram("{}/grad_histogram"\
                                            .format(v.name.replace(':', '_')), g)

            train_op = (tf.train.AdamOptimizer(learning_rate=decayed_lr)
                        .apply_gradients(grads_and_vars,
                                         global_step = global_step_tensor))
            merged = tf.summary.merge_all()

        return point, y_true, partial_loss, train_op, predictions,\
               global_step_tensor, merged, y_hat, chi
```

Kod 3: nn-shared/models.py

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

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

# 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, 'nn-shared')
CHECKMATE_DIR = os.path.join(os.path.sep, NN, 'checkmate')
sys.path.append(PYPE_DIR)
sys.path.append(CHECKMATE_DIR)

```

```python
# 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 ToyModel(Model_new.ComptonFormFactors, Model_new.ElasticDipole):
    """
    A model for CFF functions we want to learn, the rest are zero.
    """

    def __init__(self, placeholder,  CFFs=[], num_neurons_per_layers=[50,50],
                 initializer=(tf.keras.initializers
                              .VarianceScaling(scale=1.0, mode='fan_in')),
                              activation=tf.nn.selu):
        """
        Initialize the model and store placeholders for later usage.

        Args:
            placeholder (tf.placeholder): A placeholder through which the
                CFF functions acquire inputs.
            num_neurons_per_layers (list): List of numbers of neurons per
                each hidden layer.
            initializer (tf.initializer): Initializer for weights and biases.
        """
        Model_new.ComptonFormFactors.__init__(self)
        Model_new.ElasticDipole.__init__(self)
        self.input = placeholder
        assert len(num_neurons_per_layers) > 0
        self.num_neurons_per_layers = num_neurons_per_layers
        self.initializer = initializer
        self.activation = activation
        self.CFFs = CFFs

        self._NN_create()


    def _NN_create(self):
        """
        Create a neural network with user-selected CFF functions as nodes
        in the output layer.
        """

        x = layers.Dense(self.num_neurons_per_layers[0], activation=self.activation,
                         kernel_initializer=self.initializer)(self.input)

        if len(self.num_neurons_per_layers) > 1:
            for num_neurons in self.num_neurons_per_layers[1:]:
                x = layers.Dense(num_neurons, activation=self.activation,
                                 kernel_initializer=self.initializer)(x)

        if 'ImH' in self.CFFs:
            ImH_output = layers.Dense(1, kernel_initializer=self.initializer,
                                      name='ImH_output')(x)
            self._ImH = ImH_output

        if 'ReH' in self.CFFs:
            ReH_output = layers.Dense(1, kernel_initializer=self.initializer,
                                      name='ReH_output')(x)
            self._ReH = ReH_output

        if 'ImE' in self.CFFs:
            ImE_output = layers.Dense(1, kernel_initializer=self.initializer,
                                      name='ImE_output')(x)
            self._ImE = ImE_output

        if 'ReE' in self.CFFs:
```

```
            ReE_output = layers.Dense(1, kernel_initializer=self.initializer,
                                      name='ReE_output')(x)
            self._ReE = ReE_output

        if 'ImHt' in self.CFFs:
            ImHt_output = layers.Dense(1, kernel_initializer=self.initializer,
                                       name='ImHt_output')(x)
            self._ImHt = ImHt_output

        if 'ReHt' in self.CFFs:
            ReHt_output = layers.Dense(1, kernel_initializer=self.initializer,
                                       name='ReHt_output')(x)
            self._ReHt = ReHt_output

        if 'ImEt' in self.CFFs:
            ImEt_output = layers.Dense(1, kernel_initializer=self.initializer,
                                       name='ImEt_output')(x)
            self._ImEt = ImEt_output

        if 'ReEt' in self.CFFs:
            ReEt_output = layers.Dense(1, kernel_initializer=self.initializer,
                                       name='ReEt_output')(x)
            self._ReEt = ReEt_output

    def ImH(self, pt):
        """
        Calculate ImH.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ImH' in self.CFFs:
            return self._ImH
        else:
            return 0.

    def ReH(self, pt):
        """
        Calculate ReH.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ReH' in self.CFFs:
            return self._ReH
        else:
            return 0.

    def ImE(self, pt):
        """
        Calculate ImE.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ImE' in self.CFFs:
            return self._ImE
        else:
            return 0.

    def ReE(self, pt):
        """
        Calculate ReE.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ReE' in self.CFFs:
            return self._ReE
        else:
```

```
            return 0.

    def ReHt(self, pt):
        """
        Calculate ReH.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ReHt' in self.CFFs:
            return self._ReHt
        else:
            return 0.

    def ImHt(self, pt):
        """
        Calculate ImHt.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ImHt' in self.CFFs:
            return self._ImHt
        else:
            return 0.

    def ImEt(self, pt):
        """
        Calculate ImEt.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ImEt' in self.CFFs:
            return self._ImEt
        else:
            return 0.

    def ReEt(self, pt):
        """
        Calculate ImEt.
        Args:
            pt (DataPoint): Unused argument.
        """
        if 'ReEt' in self.CFFs:
            return self._ReEt
        else:
            return 0.

class Regularizer( tf.keras.regularizers.Regularizer):
    def __init__(self, strength):
        self.strength = strength
    def __call__(self, x):
        return self.strength * tf.reduce_sum(tf.square(x))

def create_graph(point_example, lr, num_neurons_per_layers,
                 loss='mse', activation='Selu',
                 reg_constant=0.0,
                 CFFs=['ImH', 'ReH', 'ReHt', 'ImHt', 'ImE', 'ReE', 'ReEt', 'ImEt']):
    """
    Create graph for learning the CFF functions.
    Args:
        point_example (DataPoint): One datapoint from some dataset for
            the placeholder to copy.
        lr (float): Learning rate.
        num_neurons_per_layers (list): List of numbers of neurons per each hidden layer.
        loss (string): 'mse' or 'huber' - defines loss function.
        activation (string): 'Selu', 'Relu' or 'Tanh'
        reg_constant (float): Regularization constant.
```

```
        CFFs (list): List of CFFs user wants to include in modeling
            the cross sections.

    Returns:
        point (tf.placeholder): Point placeholder.
        y_true (tf.placeholder): Placeholder for the true value of
            observable for a given point.
        partial_loss (tf.Tensor): Partial loss function tensor.
        train_ops (dict): A dictionary containing training operations for
            different combinations of CFF functions.
        predictions (dict): Dictionary of predictions for each CFF function.
        global_step_tensor (tf.Tensor): Global step tensor.
        merged (tf.Tensor):  Tensor of type string resulting from the merging
            of summaries tracked for visualizing in Tensorboard.
        y_hat (tf.Tensor): Tensor for observable prediction.
        chi (tf.Tensor): Tensor used for calculating \chi^2 statistic.
    """
    tf.reset_default_graph()
    with tf.device('/CPU:0'):

        with tf.name_scope('point'):
            point = copy.copy(point_example) # can be any point, I think
            # Placeholders for inputs
            point.t = tf.placeholder(tf.float32, name='t')
            point.xB = tf.placeholder(tf.float32, name='xB')
            point.y1name = tf.placeholder(tf.string, name='y1name')
            point.FTn = tf.placeholder(tf.float32, name='FTn')
            point.in1polarization = tf.placeholder(tf.float32, name='in1polarization')
            point.in2polarization = tf.placeholder(tf.float32, name='in2polarization')
            point.in2polarizationvector = tf.placeholder(tf.string, name='in2pvector')
            point.W = tf.placeholder(tf.float32, name='W')
            point.Q2 = tf.placeholder(tf.float32, name='Q2')
            point.eps = tf.placeholder(tf.float32, name='eps')
            point.yaxis = tf.placeholder(tf.string, name='yaxis')
            point.err = tf.placeholder(tf.float32, name='err')
            point.xi = tf.placeholder(tf.float32, name='xi')

        global_step_tensor = tf.train.get_or_create_global_step()
        regularizer=Regularizer(strength=reg_constant)

        with tf.name_scope('Build-model-and-theory'):
            # Create a model given the input data.
            # The same point goes to the model to calculate the CFF functions
            # and to the th.predict(point). This couldn't be solved
            # differently because the code that calculate observables
            # creates new instance of a Point and replaces my placeholders.
            # So CFF functions all have the same input ([point.t, point.xi])
            # and do not consider pt argument that is passed to them by the
            # Approach.
            if activation == 'Selu':
                act=tf.nn.selu
                init=tf.keras.initializers.VarianceScaling(scale=1.0, mode='fan_in')
            if activation == 'Relu':
                act=tf.nn.relu
                init=tf.keras.initializers.he_normal()
            if activation=='Tanh':
                act=tf.nn.tanh
                init=tf.keras.initializers.glorot_normal()

            m = ToyModel(tf.reshape(tf.stack([point.t, point.xi],0),(1,2)),
                         num_neurons_per_layers=num_neurons_per_layers,
                         CFFs=CFFs,
                         activation=act,
                         initializer=init)
            th = Approach_new.BM10tw2(m)
            th.prepare(point)
```

```
285
286         predictions = {'ImH': th.m.ImH(point),
287                        'ReH': th.m.ReH(point),
288                        'ImE': th.m.ImE(point),
289                        'ReE': th.m.ReE(point),
290                        'ImHt': th.m.ImHt(point),
291                        'ReHt': th.m.ReHt(point),
292                        'ImEt': th.m.ImEt(point),
293                        'ReEt': th.m.ReEt(point)}
294
295         with tf.name_scope('Loss-definition'):
296             y_true = tf.placeholder(tf.float32, name='y_true')
297             with tf.name_scope('y_hat'):
298                 y_hat = th.predict(point)
299
300             # Definition of loss function
301             if loss=='mse':
302                 l=tf.square(y_hat - y_true, name='squared_loss')
303             if loss=='huber':
304                 l=tf.losses.huber_loss(y_true, y_hat, delta=0.025)
305
306             #Logging weights of every defined layer, adding regularization loss term
307             reg_loss=0
308             for var in tf.global_variables():
309                 if 'kernel:0' in var.name:
310                     tf.summary.histogram("{}/weights"\
311                     .format(var.name.replace(':', '_')), var)
312                     reg_loss+=regularizer(var)
313
314             # Definition of a chi^2 value for a point passing through the graph
315             chi = tf.math.divide(tf.square(y_hat - y_true),
316                                  tf.square(point.err), name='chi')
317             #Partial loss
318             partial_loss= l + reg_loss
319
320         with tf.name_scope('Train'):
321             decayed_lr = tf.train.exponential_decay(lr,
322                                         global_step_tensor, 360,
323                                         0.97, staircase=True)
324             tf.summary.scalar('Decayed_learning_rate', decayed_lr)
325
326             #Logging layers' activations
327             n_layers=len(num_neurons_per_layers)
328             input_op=(tf.get_default_graph()
329                       .get_operation_by_name("Build-model-and-theory/Reshape"))
330             tf.summary.histogram("inputs", input_op.outputs[0])
331             for i in range(n_layers):
332                 if i == 0:
333                     op=(tf.get_default_graph()
334                         .get_operation_by_name(
335                             "Build-model-and-theory/dense/{}".format(activation)))
336                     tf.summary.histogram("Activation_{}".format(i+1), op.outputs[0])
337                 else:
338                     op=(tf.get_default_graph()
339                         .get_operation_by_name(
340                             "Build-model-and-theory/dense_{}/{}".format(i,activation)))
341                     tf.summary.histogram("Activation_{}".format(i+1), op.outputs[0])
342
343             grads_and_vars=(tf.train.AdamOptimizer(learning_rate=decayed_lr)
344                             .compute_gradients(partial_loss))
345
346             #Logging gradients of every defined layer
347             for g, v in grads_and_vars:
348                 if g is not None and 'bias' not in v.name:
349                     tf.summary.histogram("{}/grad_histogram"\
350                                         .format(v.name.replace(':', '_')), g)
```

```

            train_op = (tf.train.AdamOptimizer(learning_rate=decayed_lr)\
                        .apply_gradients(grads_and_vars,
                                         global_step = global_step_tensor))
            merged = tf.summary.merge_all()

        return point, y_true, partial_loss, train_op, predictions,\
               global_step_tensor, merged, y_hat, chi
```

Dodatak C train.py

Kod 4: 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-MC')
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 *

def create_feed(placeholder, y_true, datapoint):
    """
    Create feed from datapoint to placeholders.
    Args:
        placeholder (tf.placeholder): Datapoint
            placeholder (without val attribute).
        y_true (tf.placeholder): Placeholder for val
            attribute of the datapoint.
        datapoint (DataPoint): Datapoint to
            feed the network.
    Returns:
        feed (dict): Feed dictionary.
    """
    feed = {placeholder.in2polarization:
                float(datapoint.in2polarization),
            placeholder.t: datapoint.t,
            placeholder.xB: datapoint.xB,
            placeholder.y1name: datapoint.y1name,
            placeholder.FTn: float(datapoint.FTn),
            placeholder.in1polarization:
                float(datapoint.in1polarization),
            placeholder.in2polarizationvector:
                datapoint.in2polarizationvector
                    if hasattr(datapoint,
                               'in2polarizationvector')
```

```
                    else '',
            placeholder.W: datapoint.W,
            placeholder.Q2: datapoint.Q2,
            placeholder.eps: datapoint.eps,
            placeholder.yaxis: datapoint.yaxis,
            placeholder.err: datapoint.err,
            placeholder.xi: datapoint.xi,
            y_true: datapoint.val}
    return feed

def train(placeholder, y_true, global_step_tensor,
          merged, chi,
          partial_loss, train_op, fitpoints_train,
          fitpoints_val, save_dir, num_epochs=50,
          fine_tune_from=None, best_checkpoints_to_keep=1,
          logdir='logs'):
    """
    Train the model.
    Args:
        placeholder (tf.placeholder): Datapoint placeholder
            (without val attribute).
        y_true (tf.placeholder): Placeholder for val attribute
            of the datapoint.
        global_step_tensor (tf.Tensor): Global step tensor.
        merged (tf.Tensor):  Tensor of type string resulting from the merging
            of summaries tracked for visualizing in Tensorboard.
        chi (tf.Tensor): Tensor used for calculating \chi^2 statistic.
        partial_loss (tf.Tensor): Loss function tensor.
        train_op (tf.Op): Training operation to execute each
            step.
        fitpoints_train (list): List of training points.
        fitpoints_val (list): List of points for valudation.
        save_dir (str): Path to the directory where the
            checkpoints will be saved.
        num_epochs (int): Number of epochs to train.
        fine_tune_from (str): Path to the directory of the
            checkpoint from which to fine tune.
        best_checkpoints_to_keep (int): Number of best
            checkpoints on validation set to keep.
        logdir (str): Path to the directory to where to
            save the logs and tensorboard data.
    Returns:
        losses (list): List of losses on training set.
        val_losses (list): List of losses on validation set.
        chi2 (list): List of \chi^2/npts values on training set.
        val_chi2 (list): List of \chi^2/npts values on validation set.
    """
    if os.path.exists(save_dir):
        if len(os.listdir(save_dir)) != 0:
            raise AssertionError('save_dir should be empty!')

    saver = checkmate.BestCheckpointSaver(
      save_dir=save_dir,
      num_to_keep=best_checkpoints_to_keep,
      maximize=False
    )
    loader = tf.train.Saver()

    losses, val_losses = list(), list()
    chi2, val_chi2 = list(), list()
    rls=list()
    with tf.Session() as sess:

        sess.run(tf.global_variables_initializer())
        if fine_tune_from is not None:
            loader.restore(sess, checkmate.get_best_checkpoint(
```

```
                fine_tune_from, select_maximum_value=False,
                index=0))

        writer = tf.summary.FileWriter(logdir, sess.graph)

        for epoch in range(num_epochs):
            # Training
            for pt in fitpoints_train :
                feed = create_feed(placeholder, y_true, pt)
                _ , summary = sess.run([train_op, merged], feed_dict=feed)

            loss = 0
            chis = 0
            for pt in fitpoints_train :
                feed = create_feed(placeholder, y_true, pt)
                pl, c = sess.run([partial_loss, chi], feed_dict=feed)
                loss += pl
                chis += c
            chis/=len(fitpoints_train)

            loss_val = 0
            chi_val = 0
            for pt in fitpoints_val :
                feed = create_feed(placeholder, y_true, pt)
                pl, c = sess.run([partial_loss, chi], feed_dict=feed)
                loss_val += pl
                chi_val += c
            chi_val/=len(fitpoints_val)

            print ("{}. epoch: loss = {}".format(epoch, loss))
            losses.append(loss)
            print ("{}. epoch: chi2 = {}".format(epoch, chis))
            chi2.append(chis)
            print ("\t Loss on validation set: {}.".format(loss_val))
            val_losses.append(loss_val)
            print ("\t Chi2 on validation set: {}.".format(chi_val))
            val_chi2.append(chi_val)

            saver.handle(loss_val, sess,  global_step_tensor)
            writer.add_summary(summary, epoch)
        writer.close()
        return losses, val_losses, chi2, val_chi2
```

Dodatak D visualize_CFFs.py

Kod 5: visualize_CFFs.py

```
import numpy as np
from plotly.subplots import make_subplots
import plotly.graph_objects as go

def filtered(model,kin_var,value):
    filt=[]
    if kin_var=='x_B':
        for i,xb in enumerate(model[:,0]):
            if xb==value:
                filt.append(model[i,:])
        filt = np.array(filt)
    if kin_var=='t':
        for i,t in enumerate(model[:,1]):
            if t==value:
                filt.append(model[i,:])
        filt = np.array(filt)
```

```
    return filt

def plotCFFs(mode='lines', models=[], cffs=[], x_B=0, t=0, names=[]):
    if mode=='lines':
        colors=['crimson', 'mediumblue', 'green','darkorchid']
        names=names

        for cff in cffs:
            fig = make_subplots(rows=2, cols=2,
                                vertical_spacing=0.03, horizontal_spacing=0.03,
                                shared_xaxes=True, shared_yaxes=True,
                                subplot_titles=[r'$\xi={}$'\
                                                .format(round(x_B/(2-x_B),2)),
                                                r'$t={{{v}}}\textrm{{{GeV}}}^2$'\
                                                .format(v=t,GeV='GeV'),'','']
            for i in range(4):
                if i<2:
                    c=i+1
                    r=1
                else:
                    c=i-1
                    r=2
                dic={'H':1+r,'E':3+r,'Ht':5+r, 'Et':7+r}
                for j,model in enumerate(models):
                    if i==0 or i==2:
                        if i==0:
                            l=True
                        else:
                            l=False
                        fig.add_trace(go.Scatter(
                             x=sorted(-filtered(model,'x_B', x_B)[:,1]),
                             y=[v for _,v in sorted(zip(-filtered(model,'x_B', x_B)[:,1],
                                                filtered(model,'x_B', x_B)[:,dic[cff]]))],
                             mode='lines',
                             line=dict(color=colors[j]),name=names[j], showlegend=l),
                             row=r, col=c)
                    else:
                        xis=filtered(model,'t', t)[:,0]
                        fig.add_trace(go.Scatter(
                             x=np.array(sorted(xis))/(2-np.array(sorted(xis))),
                             y=[v for _,v in sorted(zip(filtered(model,'t', t)[:,0],
                                                filtered(model,'t', t)[:,dic[cff]]))],
                             mode='lines',
                             line=dict(color=colors[j]),showlegend=False),
                             row=r, col=c)

            fig.update_xaxes(title_text=r'$\xi$', row=2, col=2)
            fig.update_xaxes(title_text=r'$-t [\textrm{GeV}^2]$', row=2, col=1)
            fig.update_xaxes(showline=True, linewidth=1,
                             linecolor='rgba(0, 0, 0, 0.2)', row=1, col=1)
            fig.update_xaxes(showline=True, linewidth=1,
                             linecolor='rgba(0, 0, 0, 0.2)', row=1, col=2)
            fig.update_xaxes(showline=True, linewidth=1,
                             linecolor='rgba(0, 0, 0, 0.2)', row=2, col=1)
            fig.update_xaxes(showline=True, linewidth=1,
                             linecolor='rgba(0, 0, 0, 0.2)', row=2, col=2)

            if cff=='H' or cff=='E':
                fig.update_yaxes(title_text=r'$\operatorname{{{Im}}}\mathcal{{{c}}}$'\
                                 .format(Im='Im',c=cff), row=1, col=1)
                fig.update_yaxes(title_text=r'$\operatorname{{{Re}}}\mathcal{{{c}}}$'\
                                 .format(Re='Re',c=cff), row=2, col=1)
            else:
                fig.update_yaxes(title_text=r'$\operatorname{{{Im}}}\mathcal{{{c}}}$'\
                                 .format(Im='Im',c=r'\tilde{{{k}}}'.format(k=cff[0])),
                                 row=1, col=1)
```

```
                fig.update_yaxes(title_text=r'$\operatorname{{{Re}}}\mathcal{{{c}}}$'\
                                 .format(Re='Re',c=r'\tilde{{{k}}}'.format(k=cff[0])),
                                 row=2, col=1)

            fig.update_yaxes(showline=True, linewidth=1,
                             linecolor='rgba(0, 0, 0, 0.2)', row=1, col=1)
            fig.update_yaxes(showline=True, linewidth=1,
                             linecolor='rgba(0, 0, 0, 0.2)', row=1, col=2)
            fig.update_yaxes(showline=True, linewidth=1,
                             linecolor='rgba(0, 0, 0, 0.2)', row=2, col=1)
            fig.update_yaxes(showline=True, linewidth=1,
                             linecolor='rgba(0, 0, 0, 0.2)', row=2, col=2)
            fig.update_annotations(y=1.04, selector={'text':r'$\xi={}$'\
                                                     .format(round(x_B/(2-x_B),2))})
            fig.update_annotations(y=1.04,
                                   selector={'text':r'$t={{{v}}}\textrm{{{GeV}}}^2$'\
                                                     .format(v=t,GeV='GeV')})
            fig.update_layout(height=800, width=1000, template='plotly_white')
            fig.show()

    if mode=='bands':

        models1=[]
        models2=[]

        for model in models:
            models1.append(filtered(model,'x_B', x_B))
            models2.append(filtered(model,'t', t))
        models1=np.array(models1)
        models2=np.array(models2)

        ts=models1[1,:,1]
        xis=models2[1,:,0]

        avgs1=np.mean(models1[:,:,2:],axis=0)
        avgs2=np.mean(models2[:,:,2:],axis=0)
        stddevs1=np.std(models1[:,:,2:],axis=0)
        stddevs2=np.std(models2[:,:,2:],axis=0)

        for cff in cffs:
            fig = make_subplots(rows=2, cols=2,
                                vertical_spacing=0.03, horizontal_spacing=0.03,
                                shared_xaxes=True, shared_yaxes=True,
                                subplot_titles=[r'$\xi={}$'\
                                                .format(round(x_B/(2-x_B),2)),
                                                r'$t={{v}}}\textrm{{{GeV}}}^2$'\
                                                .format(v=t,GeV='GeV'),'','']
            for i in range(4):
                if i<2:
                    c=i+1
                    r=1
                else:
                    c=i-1
                    r=2
                dic={'H':r-1,'E':1+r,'Ht':3+r, 'Et':5+r}
                if i==0 or i==2:
                    if i==0:
                        l=True
                    else:
                        l=False
                    fig.add_trace(go.Scatter(
                         x=sorted(-ts),
                         y=[v for _,v in sorted(zip(-ts,
                                                np.array(avgs1+stddevs1)[:,dic[cff]]))],
                         mode='lines',
                         marker=dict(color='blue'),
```

```
                        name='Upper bound',
                        line=dict(width=2),
                        showlegend=False),
                        row=r,col=c)
                    fig.add_trace(go.Scatter(
                        x=sorted(-ts),
                        y=[v for _,v in sorted(zip(-ts,
                                                np.array(avgs1-stddevs1)[:,dic[cff]]))],
                        mode='lines',
                        marker=dict(color='mediumblue'),
                        name='Lower bound',
                        line=dict(width=2),
                        fill='tonexty',
                        fillcolor='rgba(0,90,150,0.3)',
                        showlegend=False),
                        row=r, col=c)

                else:
                    fig.add_trace(go.Scatter(
                        x=sorted(xis),
                        y=[v for _,v in sorted(zip(xis,
                                                np.array(avgs2+stddevs2)[:,dic[cff]]))],
                        mode='lines',
                        marker=dict(color='mediumblue'),
                        name='Upper bound',
                        line=dict(width=2),
                        showlegend=False),
                        row=r, col=c)
                    fig.add_trace(go.Scatter(
                        x=sorted(xis),
                        y=[v for _,v in sorted(zip(xis,
                                                np.array(avgs2-stddevs2)[:,dic[cff]]))],
                        mode='lines',
                        marker=dict(color='mediumblue'),
                        name='Lower bound',
                        line=dict(width=2),
                        fillcolor='rgba(0, 90, 150, 0.3)',
                        fill='tonexty',
                        showlegend=False),
                        row=r, col=c)

            fig.update_xaxes(title_text=r'$\xi$', row=2, col=2)
            fig.update_xaxes(title_text=r'$-t [\textrm{GeV}^2]$', row=2, col=1)
            fig.update_xaxes(showline=True,
                             linewidth=1, linecolor='rgba(0, 0, 0, 0.2)', row=1, col=1)
            fig.update_xaxes(showline=True,
                             linewidth=1,linecolor='rgba(0, 0, 0, 0.2)', row=1, col=2)
            fig.update_xaxes(showline=True,
                             linewidth=1, linecolor='rgba(0, 0, 0, 0.2)', row=2, col=1)
            fig.update_xaxes(showline=True,
                             linewidth=1,linecolor='rgba(0, 0, 0, 0.2)', row=2, col=2)

            if cff=='H' or cff=='E':
                fig.update_yaxes(title_text=r'$\operatorname{{{Im}}}\mathcal{{{c}}}$'\
                                 .format(Im='Im',c=cff), row=1, col=1)
                fig.update_yaxes(title_text=r'$\operatorname{{{Re}}}\mathcal{{{c}}}$'\
                                 .format(Re='Re',c=cff), row=2, col=1)
            else:
                fig.update_yaxes(title_text=r'$\operatorname{{{Im}}}\mathcal{{{c}}}$'\
                                 .format(Im='Im',c=r'\tilde{{{k}}}'.format(k=cff[0])),
                                 row=1, col=1)
                fig.update_yaxes(title_text=r'$\operatorname{{{Re}}}\mathcal{{{c}}}$'\
                                 .format(Re='Re',c=r'\tilde{{{k}}}'.format(k=cff[0])),
                                 row=2, col=1)

            fig.update_yaxes(showline=True, linewidth=1,
```

```
                                linecolor='rgba(0, 0, 0, 0.2)', row=1, col=1)
            fig.update_yaxes(showline=True, linewidth=1,
                                linecolor='rgba(0, 0, 0, 0.2)', row=1, col=2)
            fig.update_yaxes(showline=True, linewidth=1,
                                linecolor='rgba(0, 0, 0, 0.2)', row=2, col=1)
            fig.update_yaxes(showline=True, linewidth=1,
                                linecolor='rgba(0, 0, 0, 0.2)', row=2, col=2)

            fig.update_annotations(y=1.04,
                                   selector={'text':r'$\xi={}$'\
                                   .format(round(x_B/(2-x_B),2))})
            fig.update_annotations(y=1.04,
                                   selector={'text':r'$t={{{v}}}\textrm{{{GeV}}}^2$'\
                                   .format(v=t,GeV='GeV')})

            fig.update_layout(height=800, width=1000, template='plotly_white')
            fig.show()
```

Literatura

[1] Thomson, M. Modern Particle Physics. Cambridge: Cambridge University Press, 2013.

[2] A.V. Belitsky; A.V. Radyushkin Unraveling hadron structure with generalized parton distributions. Phys.Rept. 418 (2005).

[3] Georges, F. Deeply virtual Compton scattering at Jefferson Lab. Doktorski rad. Université Paris-Saclay, 2018.

[4] Kumerički, K.; Liuti S.; Moutarde H. GPD phenomenology and DVCS fitting - Entering the high-precision era. Eur. Phys. J. A 52, 157 (2016).

[5] Belitsky A.V.; Müller D.; Kirchner A. Theory of deeply virtual Compton scattering on the nucleon. Nucl.Phys.B 629, 323 (2002).

[6] Neural Network Terminology `https://cs231n.github.io/neural-networks-1/`, [10.09.2020.]

[7] From Perceptron to Deep Neural Nets `https://becominghuman.ai/from-perceptron-to-deep-neural-nets-504b8ff616e`, [10.09.2020.]

[8] Introduction to Activation Functions `https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092`, [10.09.2020.]

[9] Funahashi, K.I. On the approximate realization of continuous mappings by neural networks. Neural Networks, Volume 2, Issue 3, 183-192. (1989).

[10] Bishop, Christopher M. Pattern Recognition and Machine Learning. New York :Springer, 2006.

[11] Dalbelo Bašić B.; Čupić M.; Šnajder J. Umjetne neuronske mreže. Fakultet elektrotehnike i računarstva, 2008.

[12] Optimization in Deep Learning `https://dragonnotes.org/DeepLearning/Optimization`, [10.09.2020.]

[13] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. Learning representations by back-propagating errors. Nature. 323 (6088): 533–536 (1986a).

[14] Learning Process of a Neural Network https://towardsdatascience.com/how-do-artificial-neural-networks-learn-773e46399fc7, [10.09.2020.]

[15] Activation Functions https://codeodysseys.com/posts/activation-functions/7, [12.09.2020.]

[16] Glorot, X.; Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. Journal of Machine Learning Research - Proceedings Track. 9. 249-256 (2010).

[17] He K.; Zhang X.; Ren S.; Sun J. Deep Residual Learning for Image Recognition. Computer Vision and Pattern Recognition (CVPR), 2016.

[18] Ćorić I. Istraživanje kvarkovsko-gluonske strukture protona pomoću strojnog učenja. Diplomski rad. Zagreb : Prirodoslovno-matematički fakultet, 2019.

[19] Cvitković M. Razotkrivanje strukture nukleona pomoću neuronskih mreža. Samostalni seminar. Zagreb : Prirodoslovno-matematički fakultet, 2020.

[20] Goloskokov S. V.; Kroll P. The role of the quark and gluon GPDs in hard vector-meson electro-production. Eur. Phys. J. C53 , 2008, 367–384

[21] Kumerički K.; Müller D.; Schäfer A. Parametrizing Compton form factors with neural networks. Nuclear Physics B Vol 222-224 (2011).

[22] Čuić M.; Kumerički K.; Schäfer A. Separation of Quark Flavors using DVCS Data. (2020).

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

[24] Elsayed N.; Maida A.; Bayoumi M. Effects of Different Activation Functions for Unsupervised Convolutional LSTM Spatiotemporal Learning. Advances in Science, Technology and Engineering Systems Journal. (2019).

[25] Huber Loss https://en.wikipedia.org/wiki/Huber_loss, [12.09.2020.]

[26] Checkmate https://github.com/vonclites/checkmate