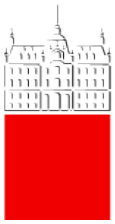
Univerza *v Ljubljani*
Fakulteta za *matematiko in fiziko*



Strojno učenje

12. naloga pri Matematično-fizikalnem praktikumu

Avtor: Marko Urbanč (28191096)
Predavatelj: prof. dr. Borut Paul Kerševan

8.9.2023

Kazalo

1 Uvod

Medtem ko se je še nedavno zdelo strojno učenje prava temna magija (vsaj meni), je pravzaprav dandanes uporaba različnih algoritmov strojnega učenja popolnoma vsakdanja in že rutinska. Ljudje se pravzaprav stalno srečujemo z različnimi algoritmi strojnega učenja, ki nam pomagajo pri različnih stvareh, kot so npr. priporočila na Netflixu, Googlovi iskalni algoritmi, različni algoritmi za prepoznavanje objektov na slikah, itd. Vse to so algoritmi, ki so zasnovani na strojnem učenju.

Prvotno sem želel pustiti prejšnji stavek brez pojasnila, vendar sem se odločil, da bom vseeno napisal nekaj besed o kar mislim. Poglejmo prvo Netflix. Netflix je spletna storitev za ogled filmov in serij. Ker je na Netflixu ogromno filmov in serij, je težko najti tisto, kar bi si želeli ogledati. Zato Netflix uporablja algoritem, ki na podlagi vaših prejšnjih ogledov in ocen priporoča filme in serije, ki bi vam lahko bili všeč [1]. Ampak ne samo to, glede na to iz katere naprave gledate, ob kakšnem času dneva in drugih parametrih, ki jim Netflix pravi Kontekst [2], pravzaprav so pa to *contextual bandits*, ki se uporabljajo v enem okusu strojnega učenja. Več o tem kasneje. V kolikor sem uspel razumeti ta članek zgleda, da še več kot to, želijo praktično *in real time* prilagajati priporočila, glede na to kakšen je vaš t.i. *intent* [3]. Oh in vsi, ki si med seboj delite Netflix račune, nikakor ne skrbite, tudi to vas zna nekoč tepst kakšen algoritem strojnega učenja [4].

Nadalje, Google. Google je spletni iskalnik, ki ga praktično vsi uporabljamo. Google uporablja algoritme strojnega že precej dolgo časa. Leta 2015 so predstavili deep learning sistem RankBrain, ki je bil namenjen izboljšanju rezultatov iskanja. RankBrain je bil namenjen predvsem iskanju poizvedb, ki jih Google še nikoli ni videl. RankBrain je bil zasnovan tako, da je na podlagi preteklih iskanj in rezultatov iskanj, ki so jih uporabniki izbrali, izboljševal rezultate iskanja. Od 2018 naprej so v Google Search v rabi nevronske mreže, ki so namenjene predvsem izboljšanju rezultatov iskanja. Od 2019 naprej pa Google uporablja tudi BERT [5], ki je bil ogromen korak naprej v razumevanju naravnega jezika.

Na hitro o tipih strojnega učenja oz. osnovnih vrstah algoritmov strojnega učenja. Poznamo

- Nadzorovano učenje (supervised learning)
  - Klasifikacija (classification): Sortiranje podatkov v razrede. Primer: Razpoznavanje številk na sliki.
  - Regresija (regression): Modeliranje odvisnosti med podatki. Primer: Napovedovanje cene nepremičnine.
- Nenadzorovano učenje (unsupervised learning)
- Stimulirano učenje (reinforcement learning)

Poglejmo si zdaj uporabo strojnega učenja v fizikalnem kontekstu. V fiziki se strojno učenje uporablja za različne namene. Največkrat uporabljamo algoritme prvega tipa, torej jih nadzorovano učimo. V fiziki visokih energij se

strojno učenje uporablja za razpoznavanje delcev v detektorjih, za razpoznavanje različnih pojavov, za izboljšanje različnih meritev, itd. To je nekaj kar bomo tudi sami počeli v tej nalogi. Iskali bomo Higgsov bozon v podatkih, ki jih je zbrala ATLAS kolaboracija.

1.1 Kako delujejo algoritmi strojnega učenja?

Pred tem, pa še osnovno o tem, kako sploh delujejo algoritmi strojnega učenja. Imamo nabor parametrov $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{N}$, kjer je $x_k = (x_k^1, ...x_k^M)$ naključno izbrani vektor z M lastnostmi oz. *karakteristikami*, $y_k = (y_k^1, ..., y_k^Q)$ pa je vektor Q ciljnih vrednosti, ki so lahko diskretne ali realna števila ali kaj drugega (npr. barve, ki jim tudi lahko priredimo številske vrednosti). Vrednosti (x_k, y_k) so neodvisne in porazdeljene po neznani porazdelitvi $\mathcal{P}(\cdot, \cdot)$. Cilj strojnega učenja je poiskati oz. priučiti preslikavo $h : \mathbb{R}^Q \to \mathbb{R}$, ki bo minimizirala pričakovano vrednost funkcije izgube (angl. loss function) $\mathcal{L}(h)$

$$\mathcal{L}(h) = \mathbb{E}\mathrm{L}(\vec{y}, \vec{h}(\vec{x})) = \frac{1}{N}\sum_{k=1}^{N} \mathrm{L}(y_k, \vec{h}(x_k)) \,, \tag{1}$$

kjer je $\mathrm{L}(\cdot, \cdot)$ gladka funkcija, ki opisuje oceno za kvaliteto napovedi, pri čemer so $(\vec{x}, \vec{y})$ neodvisno vzorčene iz nabora $\mathcal{D}$ po porazdelitvi $\mathcal{P}(\cdot, \cdot)$. Po koncu učenja imamo torej na voljo funkcijo $h(\vec{x})$, ki nam za nek vhodni nabor vrednosti $\hat{x}$ poda napoved $\hat{y} = \vec{h}(\hat{x})$, ki ustrezno kategorizira ta nabor vrednosti.

Funkcije $\vec{h}$ so v praksi sestavljene iz množice preprostih funkcij z prostimi parametri, kar na koncu seveda pomeni velik skupni nabor neznanih parametrov in zahteven postopek postopek minimizacije funkcije izgube.

1.2 Odločitvena drevesa (Decision trees)

Odločitvena drevesa so ena izmed najbolj preprostih metod strojnega učenja. Odločitvena drevesa so sestavljena iz vozlišč in povezav. Vozlišča so povezana z povezavami, ki so lahko usmerjene ali neusmerjene. Osnovni gradnik odločitvenega drevesa je kar stopničasta funkcija $\mathrm{H}(x_i, -t_i) = 0, 1$, ki je enaka ena za $x_i > t_i$ in nič za $x_i < t_i$, kjer je x_i ena izmed karakteristik in t_i neznani parameter. Iz skupine takšnih funkcij, ki predstavljajo binarne odločitve lahko skonstruiramo končno uteženo funkcijo

$$\vec{h} = \sum_{i=1}^{J} \vec{w}_i \mathrm{H}(x_i, -t_i) \,, \tag{2}$$

kjer so $\vec{w}_i$ vektorji neznanih uteži. Tako t_i kot $\vec{w}_i$ lahko določimo, v procesu učenja. Nadgradnjo odločitvenih dreves predstavljajo pospešena odločitvena drevesa (angl. boosted decision trees), ki uporabljajo več odločitvenih dreves, ki so med seboj povezana. Temeljijo pa na Gradient Boosting algoritmu [6], ki je bil razvit leta 1999 in je bil namenjen izboljšanju napovedi. Gradient Boosting algoritem je bil razvit za regresijske probleme, vendar se ga da uporabiti tudi za klasifikacijske probleme in je v praksi zelo uspešen.

1.3 Nevronske mreže (Neural networks)

Pri nevronskih mrežah gre za to, da imamo množico nevronov, ki so med seboj povezani. Vsak nevron je povezan z vsakim nevronom v naslednjem sloju. Nevroni so med seboj povezani z utežmi, ki jih lahko spreminjamo. Slojev je lahko več, odvisno od tega, kako kompleksen model želimo. Poznamo vidne sloje, skrite sloje in izhodne sloje. Nevroni v vidnem sloju so povezani z vhodnimi podatki, nevroni v skritih slojih so žal za nas neke sorte *black box*, ki ga ne moremo razumeti, izhodni sloj pa nam poda napoved. Nevronske mreže so zelo uspešne pri klasifikaciji in regresiji. Obstaja tudi cel kup podvrst nevronskih mrež, kot so npr. konvolucijske nevronske mreže, ki so zelo uspešne pri razpoznavanju objektov na slikah, rekurentne nevronske mreže, ki so zelo uspešne pri razpoznavanju govora, itd.

Osnoven gradnik nevronskih mrež je *perceptron*, ki ga opisuje funkcija

$$\mathrm{h}_{w,b} = \theta(\vec{w}^{\intercal} \cdot \vec{X} + b)\,, \tag{3}$$

kjer je $\vec{w}$ vektor uteži, $\vec{X}$ vektor vhodnih vrednosti, b pa je neka konstanta, ki ji pravimo *bias*. Preko tega tvorimo uteženo vsoto. Funkcije θ se imenuje *aktivacijska funkcija*. Najbolj pogosta je *sigmoidna* funkcija, ki je definirana kot

$$\theta(x) = \frac{1}{1 + e^{-x}}\,. \tag{4}$$

Ta vpelje neke vrste nelinearnost v model. Če bi uporabili linearno funkcijo, bi namreč dobili linearno preslikavo. Dandanes se sicer poleg sigmoidne funkcije uporabljajo tudi druge funkcije, verjetno je najbolj znana *ReLU* funkcija [7], ki je definirana kot

$$\theta(x) = \max(0, x)\,. \tag{5}$$

Ta je boljša zaradi več razlogov, ampak v glavnem so prednosti, da se jo bistveno hitreje računa in da je bolj odporna na *vanishing gradient* problem. Nevronske mreže se učijo preko dveh ključnih algoritmov in sicer *back propagation* in *gradient descent*. Back propagation je algoritem, ki nam omogoča, da izračunamo gradient funkcije izgube po vseh utežeh v nevronski mreži. S tem lahko popravimo uteži v nevronski mreži za en korak. Gradient descent pa je algoritem, ki nam omogoča, da se sprehodimo po funkciji izgube in poiščemo lokalni minimum. Problem, ki ga ima recimo sigmoida je, da ima majhno vrednost gradienta, drugod pa se funkcija asimptotsko obnaša in je gradient tam enak nič. Kar se zgodi je, da se zniža vrednost gradienta za vsak sloj, ki ga imamo v nevronski mreži. To pomeni, da se bodo prvi sloji nepravilno učili. Rečeno malo drugače, njihov gradient izgine in ker se to zgodi, bo tudi aktivacijska funkcija premikala vrednost proti nič. Z uporabo ReLU funkcije tega problema nimamo, saj je gradient konstanten in ne izgine. To pomeni, da nam modeli v splošnem hitreje konvergirajo. Seveda pa tudi ReLU funkcija ni brez svojih napak. Pojavi se lahko ravno kontra pojav, torej *exploding gradient*. Obstaja potem cela vrsta izboljšav, da bi se temu izognili.

Nevronska mreža je sestavljena iz poljubne topologije prej omenjenih perceptronov, ki na začetku sprejmejo karakteristiko dogodka $\vec{x}$ v končni fazi pa vrnejo

napoved $\hat{y}$, ki mora biti lim bližje pravi vrednosti y. Z uporabo ustrezne funkcije izgube, recimo MSE, definiran kot

$$\text{MSE} = \mathcal{L}(h) = \mathbb{E}||\vec{y} - \hat{y}||^2 \,, \tag{6}$$

se problem znova prevede na minimizacijo, kjer znova iščemo minimum funkcije izgube, za neznan nabor uteži $\vec{w}$ in konstant b_i. Globoke nevronske mreže niso nič drugega, kot velike nevronske mreže, ali skupine le-teh.

2 Naloga

Naloga nam tokrat poda ves material potreben, tako kodo, kot vzorce, za ločevanje dogodkov Higgsovega bozona od ostalih procesov ozadja. V naboru simuliranih dogodkov je 18 karakteristik, katerih vsaka posamezno zelo slabo loči signal od ozadja. Preko ML pa lahko dosežemo kar dober uspeh.

Naloga zahteva od nas da za obe metodi, tako BDT kot NN, določimo uspešnost in narišemo ROC krivuljo, za nekaj tipičnih konfiguracij. Naj bomo pozorni na vpliv vhodnih parametrov na uspešnost in na število uporabljenih spremenljivk.

3 Opis reševanja

Reševanja sem se lotil tako, da sem si najprej malo ogledal profesorjevo kodo. Kmalu sem ugotovil, da despite komentarjem in kakšnega docstringa v kodi, bo za razumevanje šlo hitreje če si sam napišem programe. As usual je Python standardno orodje z bibliotekami `numpy`, `pandas`, `matplotlib`, `scikit-learn`, `catboost` in `torch`. Sparking the debate oz. tekmovalnost med `tensorflow`, ki ga je uporabil profesor in `pytorch`, ki ga imam jaz raje.

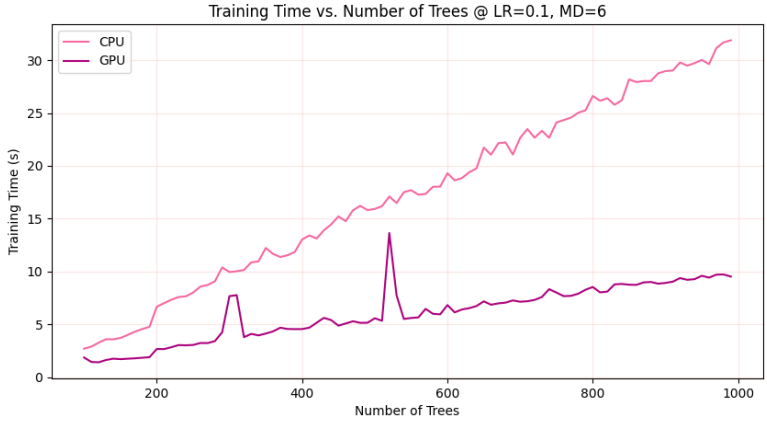
Kot je zdaj navada sem si napisal nekaj razredov za lažje delo. Začel sem z razredom za BDT modele, ki sem ga poimenoval `BoostMe`. V njem sem definiral metode za učenje modela, za napovedovanje in za izris ROC krivulje. Za učenje sem uporabljal `catboost` knjižnico, bolj specifično `CatBoostClassifier`. Se mi zdi, da sem kar navdušen nad njo.

Nato sem napisal razred za NN modele, ki sem ga poimenoval `WhereArtThouHiggs`. V njem sem definiral metode za ustvarjenje modela, za učenje modela, za napovedovanje in za izris ROC krivulje. Za ustvarjanje modela sem uporabil `torch` knjižnico, bolj specifično `torch.nn.Sequential`, ki sem ga napolnil z tremi sloji `torch.nn.Linear` in aktivacijsko funkcijo `torch.nn.ReLU`. Zadnji sloj ima Sigmoidno aktivacijsko funkcijo. `torch.nn.Sigmoid`. To mi je malo smešno delalo, tako da sem kasneje zamenjal na svoj model, s tem da sem napisal razred `Net`, ki je podrazred `torch.nn.Module` in ga napolnil z prej omenjenimi sloji. Število skritih slojev je bilo tako 3 nevronov v njih pa 100. Za učenje sem uporabil **Stochastic gradient descent** `torch.optim.SGD` in kot smo rekli v uvodu kar `torch.nn.MSELoss` za funkcijo izgube.

4 Rezultati

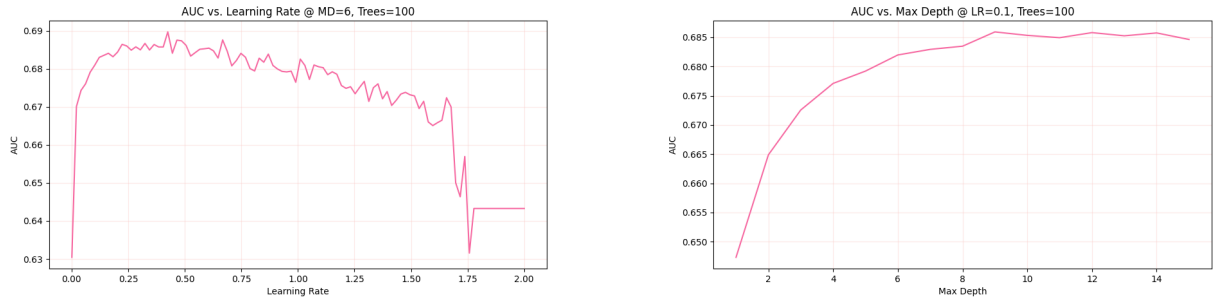
4.1 BDT

Glavni parameter, ki ga lahko spreminjamo pri algoritmu BDT je število dreves, ki jih uporabimo. Začel sem z 100 drevesi in nato število dreves povečeval. Logično sledi obvezna primerjava med časom izračunov na GPU in na CPU. GPU je RTX 3060Ti, CPU pa Ryzen 5 3600X. Rezultati so prikazani na sliki 1.



Slika 1: Čas izračuna BDT modela na GPU in CPU.

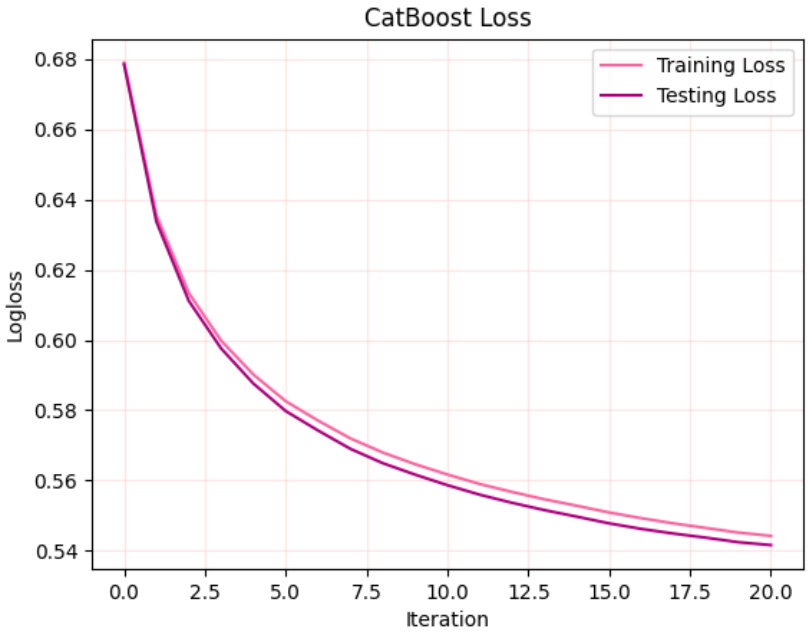
Kot vidimo je čas izračuna na GPU veliko hitrejši. Čeprav je to pričakovano, je vseeno zanimivo videti, da je razlika v času izračuna kar precejšnja. Zdaj lahko vsi prepričujemo svoje starše, da potrebujemo grafično kartico za študij. Smiselno je pogledati še druga dva pomembna parametra in sicer *learning rate* in *max_depth*. So na sliki 2.



Slika 2: Vpliv *learning rate* in *max_depth* na uspešnost BDT modela.

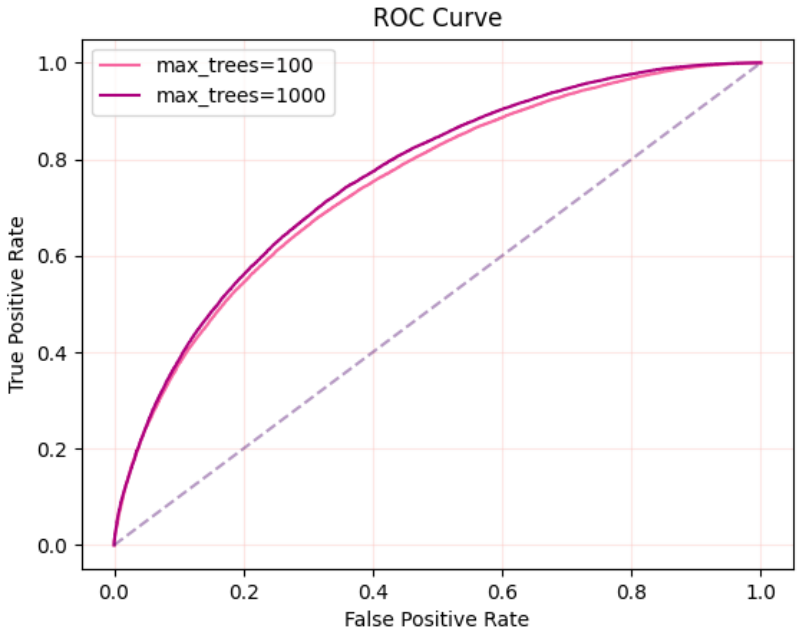
Vidimo, da ocena za uspešnost modela lepo narašča z večanjem *max_depth* in *learning rate*. Očitno v temu primeru so vse globine od 6 naprej odveč ker je rezultat relativno konstanten. *Learning rate* ima velik vpliv na uspešnost mo-

dela in vidimo da je lepo izbrati neko srednjo vrednost. Premala vrednost in bo model počasen, prevelika vrednost in bo model neuspešen. Izrisal sem pa tudi kako se vrednost funkcije izgube manjša za model z 100 drevesi, *max_depth* = 6 in *learning rate* = 0.1. Rezultat je na sliki 4.



Slika 3: ROC krivulja za BDT model z 100 drevesi, *max_depth* = 6 in *learning rate* = 0.1.

In seveda tudi ROC krivuljo za model z prej omenjenemi parametri in za model, ki ima spremenjeno število dreves na 1000. Rezultat je na sliki 4.

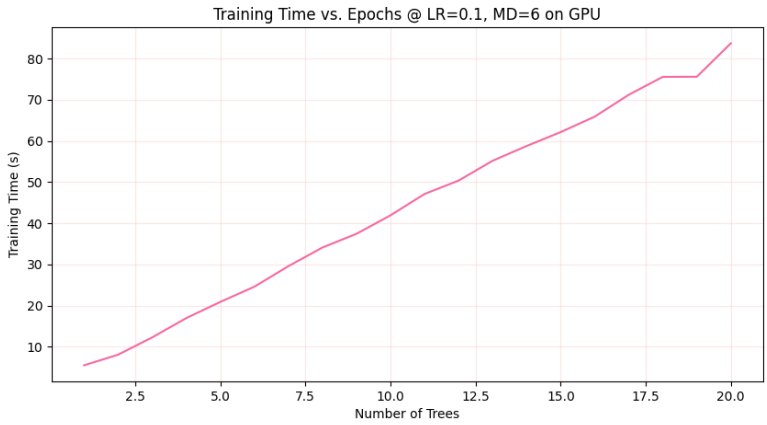


Slika 4: ROC krivulja za BDT model z 100 drevesi, *max_depth* = 6 in *learning rate* = 0.1 ter za model z 1000 drevesi, *max_depth* = 6 in *learning rate* = 0.1.

Vidimo, da je razlika med njima pravzaprav precej marginalna. Glede na to, da je prvi model porabil le 1.57 s za učenje, medtem ko je drugi porabil 10.45 s, se mi zdi da je prvi model boljša izbira, glede na majhno razliko v uspešnosti, a relativno veliko ceno v času izračuna.

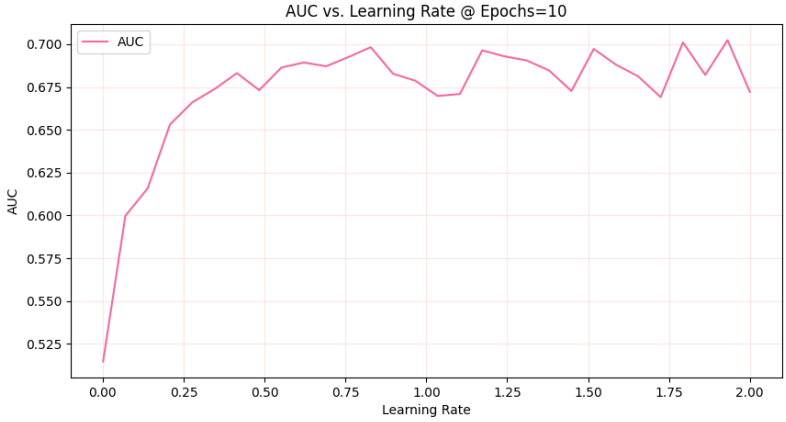
4.2 NN

V glavnem sem tu poskusil rekreirati profesorjeve rezultate. Zaradi velikosti podatkovnega seta in pa narave učenja NN modelov, traja učenje kar nekaj časa. Žal zaradi mojih lastnih napak, nimam dovolj časa, da bi se igral z vsemi parametri in bi lahko naredil kakšno boljšo analizo. Ker je že učenje z GPU trajalo dolgo, se testov z CPU sploh lotil nisem. Vseeno pa sem narisal odvisnost časa učenja od števila epoh, ki je na sliki 5.



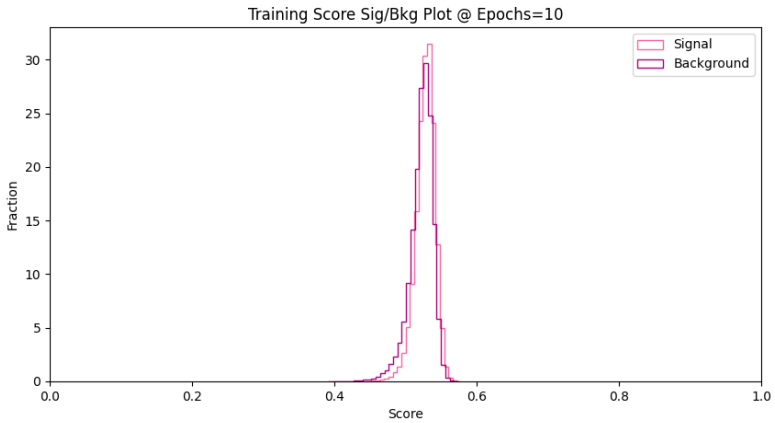
Slika 5: Čas učenja NN modela na GPU.

Vidimo, da je čas precej linearno odvisen in kar dolg, še sploh v primerjavi z hitrostjo BDT modela. Se mi zdi, da bi jaz v praksi uporabljal BDT model, za reševanje tega problema. Zaradi časovne stiske nisem uspel narediti toliko hyperparameter sweepov, kot bi si želel. Definitivno bi si želel narediti tudi študijo za vpliv števila skritih slojev in števila nevronov na uspešnost modela, ampak saj veste, čas. Sem pa vseeno med pisanjem tega poročila pognal sweep za *learning rate*, ki je poleg števila epoh verjetno še najbolj pomemben parameter.



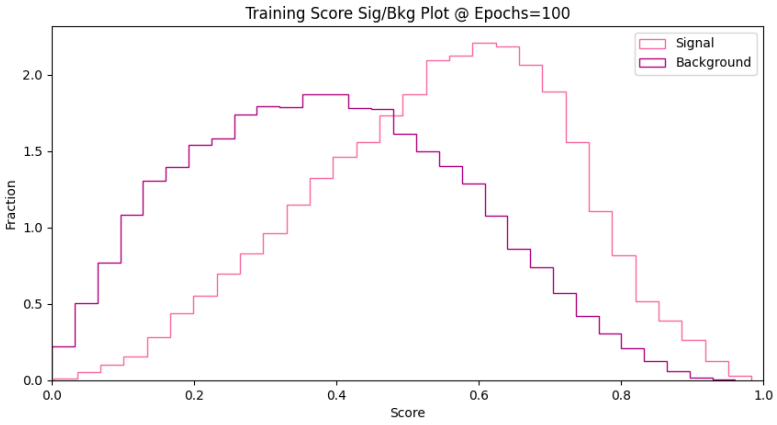
Slika 6: Vpliv *learning rate* na uspešnost NN modela.

Moram priznati, da je rezultat nekoliko nepričakovan, saj sem mislil, da bo po nekaj časa uspešnost padla, ampak se zdi, da nekako uspe ostati še kar konstantna. Verjetno je tu spet neka srednja mera najboljša izbira. Bi bilo pa zanimivo pogledati, kako to zgleda v ekstremu. Za konec poglejmo pa še uspešnost modela pri napovedih. Začel sem konzervativno in uporabil samo 10 epoh. Rezultat je na sliki 7. Tu vrednost med 0 in 1 določi, kako prepričan je naš NN, da je to kar je dobil signal Higgsovega bozona.



Slika 7: Porazdelitev izhodov za NN model z 10 epohami.

Vidimo, da pri samo 10 epohah, model še ni dovolj dober. Signal je neprepoznaven od ozadja in izbira je bolj ali manj naključna. Poglejmo še kako se model obnese pri 100 epohah. Rezultat je na sliki 8.

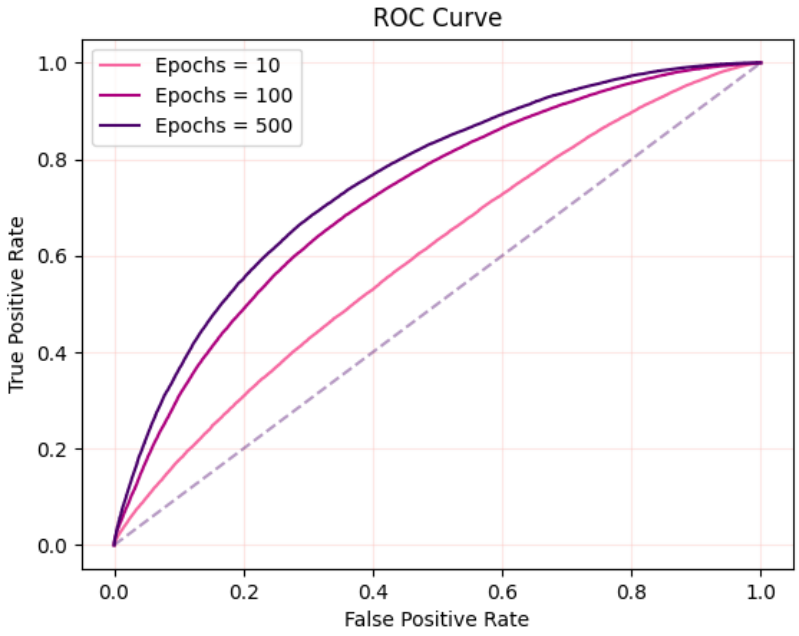


Slika 8: Porazdelitev izhodov za NN model z 100 epohami.

Tu je razlika že boljša. Moč je ločiti signal od ozadja, ampak se precej prekrivata. Namesto, da bi ta projekt oddal ob neki smiselni uri, sem se odločil potegniti pozno v noč in pognati še 500 epoh. Rezultat je na sliki 9.

Slika 9: Porazdelitev izhodov za NN model z 500 epohami.

To se pa zdi že kar sprejemljivo. Signal je lepo ločen od ozadja, vseeno je nekaj prekrivanja, ampak to je v redu. ROC krivulje za vse modele so na sliki 10. Kot pričakovano iz prejšnjih slik, je korak med 10 in 100 epohami veliko večji kot med 100 in 500 epohami. Želim si, da bi imel veliko več časa, da bi se s tem lahko bolj igral.



Slika 10: ROC krivulje za NN modele z 10, 100 in 500 epohami.

5 Komentarji in izboljšave

Predvsem je problem pri tej nalogi, da je ML tako čudovito področje, meni pa je zmanjkalo časa, da bi se lahko igral z njim v tem akademskem smislu. Toliko pojavov bi se dalo raziskati, recimo to kar sem omenil v uvodu *vanishing gradient* problem, *exploding gradient* problem, *overfitting*, *underfitting*, Želim si tudi, da bi lahko izrisal tudi kakšne lepše grafe dejanskih rezultatov, ker trenutno je bilo vse skupaj malo bolj abstrahirano samo na te hiperparametre (no ja za NN je potem še kar okay ratalo). Pravzaprav bi si pri vseh grafih želel kaj dodati, ampak žal mi je zmanjkalo časa. Zelo škoda, ker mi je bila verjetno ta naloga od vseh najbolj zabavna, sploh kar se tiče programerskega duha.

Hvala za vso znanje, ki sem ga pridobil profesor.

Literatura

[1] Sudarshan Lamkhede and Christoph Kofler. Recommendations and results organization in netflix search, 2021.

[2] Harald Steck, Linas Baltrunas, Ehtsham Elahi, Dawen Liang, Yves Raimond, and Justin Basilico. Deep learning for recommender systems: A netflix case study. *AI Magazine*, 42(3):7–18, Nov. 2021.

[3] Chao-Yuan Wu, Christopher V. Alvino, Alexander J. Smola, and Justin Basilico. Using navigation to improve recommendations in real-time. *Proceedings of the 10th ACM Conference on Recommender Systems*, 2016.

[4] Soheil Esmaeilzadeh, Negin Salajegheh, Amir Ziai, and Jeff Boote. Abuse and fraud detection in streaming services using heuristic-aware machine learning, 2022.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina N. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.

[6] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5), 2001.

[7] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019.