GIMNAZIJA POLJANE

RAZISKOVALNA NALOGA

UMETNA INTELIGENCA V BIOLOGIJI

GLOBOKI MODELI ZA NAPOVEDOVANJE CELIČNIH TIPOV IZ PODATKOV O GENSKIH IZRAZIH

RAČUNALNIŠTVO IN INFORMATIKA

Mentor

prof. dr. Blaž ZUPAN

Avtor

Mark D. LONGAR, 3.A

Somentor

prof. Boštjan ŽNIDARŠIČ

LJUBLJANA, FEBRUAR 2020

Zahvala

Hvala Blažu za neizmerno priložnost in znanje.

POVZETEK

Analiza podatkov o genskih izrazih posameznih celic je pomemben korak pri razumevanju celične sestave in delovanja kompleksnih tkiv. Klasični pristopi uporabljajo nenadzorovano gručenje celic in ročno razvrščanje celičnih tipov posameznih skupin. Ti klasični pristopi niso praktični pri večjih naborih podatkov, zato se je pojavila potreba po avtomatiziranih pristopih klasifikacije celičnih tipov. Sodobni pristopi uporabljajo markerske gene za prepoznavanje vrst celic. V nalogi smo razvili metodo globokega učenja, ki je sposobna nenadzorovanega, delno nadzorovanega in nadzorovanega učenja z vključitvijo povezav med markerskimi geni in njihovimi ustreznimi celičnimi tipi v strukturo nevronske mreže samokodirnika. S poskusi na že objavljenih eksperimentalnih podatkih smo ugotovili, da vključena dodatna znanja o markerskih genih pomagajo nevronskim mrežam in dirigirajo napovedno točnost.

Ključne besede: Genski izrazi, markerski geni, klasifikacija, samokodirnik

ABSTRACT

Single-cell RNA sequencing data analysis is an important step in cellular decomposition of complex tissues. Classical approaches use unsupervised clustering of cells and a manual annotation of their cell types. These classical approaches do not scale well to bigger data sets and consequentially, there has developed a need for automated approaches of cellular annotation. Contemporary approaches use marker genes to identify cell types. In this paper we develop a deep learning method capable of unsupervised, semi-supervised and supervised learning by incorporating the relationships between marker genes and their corresponding cell types into the structure of an autoencoder neural network. With experiments on already-published experimental data, we conclude that additional knowledge of marker genes helps neural networks learn and directs their classification accuracy.

Keywords: scRNA-seq, marker genes, classification, autoencoder

KAZALO

1 UVOD 1

2 UMETNA INTELIGENCA 4

2.1 STROJNO UČENJE 6

2.1.1 Načini učenja 6

2.1.2 Modeli strojnega učenja 9

2.2 NEVRONSKE MREŽE 10

2.2.1 Struktura nevronskih mrež 10

2.2.2 Učenje nevronskih mrež 12

2.3 GLOBOKO UČENJE 16

2.3.1 Samokodirniki 17

3 IDENTIFIKACIJA CELIČNIH TIPOV S SAMOKODIRNIKOM 19

3.1 METODA 20

3.1.1 Struktura modela 20

3.1.2 Priprava podatkov 22

3.2 IMPLEMENTACIJA MODELA IN EKSPERIMENTOV 24

3.2.1 Razpoložljivost podatkov 24

4 REZULTATI 25

***H_1** – Struktura nevronske mreže markerjev je ključna 26*

***H_2** – Nenadzorovani model bo bolje napovedoval celične tipe 27*

***H_3** – Delno nadzorovani model bo dosegel boljše rezultate 28*

5 ZAKLJUČEK 29

6 VIRI 30

7 LITERATURA 32

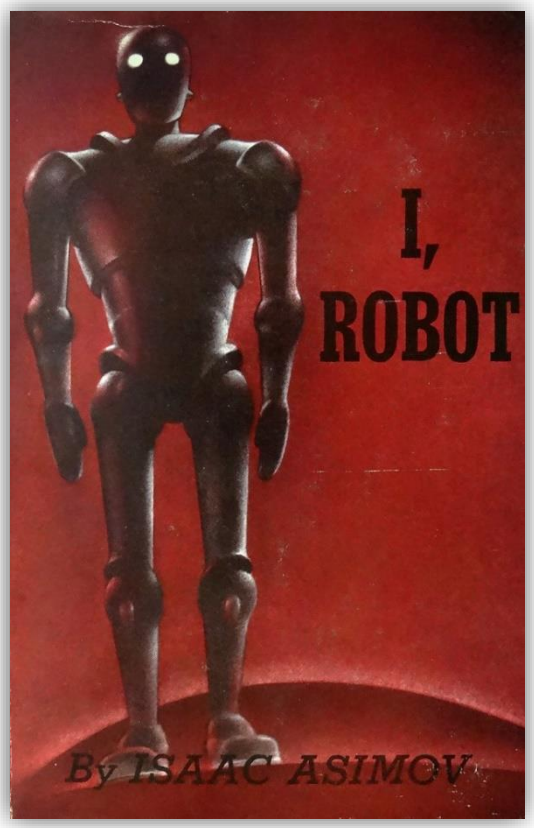
PRILOGA A 33

DEEP/ANALYSIS.PY 33

DEEP/DATA.PY 42
DEEP/MODEL.PY 45
DEEP/PREPROCESSING.PY 48
BACKEND/ANALYSIS.PY 49
BACKEND/DATA.PY 52
BACKEND/MODEL.PY 54
BACKEND/PREPROCESSING.PY 59

1 UVOD

Danes velikokrat slišimo besedno zvezo umetna inteligenca in si predstavljamo prizore iz znanstvenofantastičnih knjig, filmov in nanizank, kot so Jaz, robot (1950), 2001: Vesoljska odiseja (1968) in Black mirror (2011), ki prikazujejo distopično prihodnost človeštva s prihodom inteligentnih sistemov. Ljudje se bojijo prihodnosti, v kateri roboti zamenjajo vso delovno silo, umetna inteligenca pa vlada našemu vsakdanjemu življenju. Kljub temu se redko vprašamo o trenutnem stanju razvoja umetne inteligence, ki se danes ne zdi več taka znanstvena fantastika kot pred 10, 20 ali 100 leti.



Slika 1: Izvirna platnica knjige Jaz, robot Isaaca Asimova (1950)

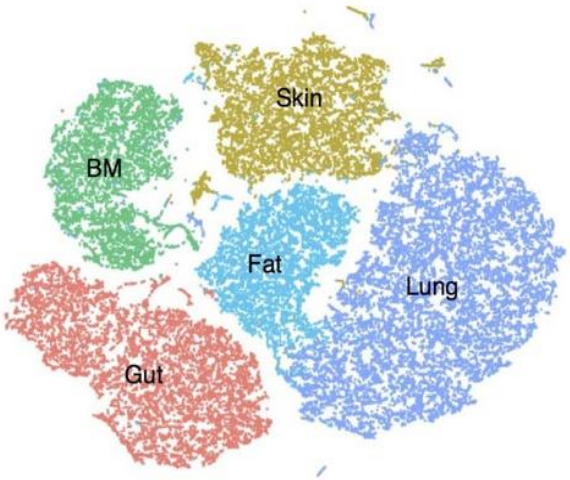
Umetno inteligenco uporabljamo vsak dan, ne da bi se tega sploh zavedali, npr. pri brskanju po družabnih omrežjih in pri nakupovanju na spletu, hkrati pa se na področjih kot sta medicina (npr. postavljanje diagnoz na podlagi rentgenskih slik prsnega koša [1]) in avtomobilska industrija (samovozeči avtomobili) hitro razvijajo nove metode, ki bi lahko neizmerno spremenile svet.

Eno izmed področij, kjer se uveljavlja uporaba umetne inteligence, je tudi *bioinformatika*. Bioinformatika je interdisciplinarna veda, ki povezuje znanje računalništva in informatike, kot sta umetna inteligenca in *podatkovno rudarjenje*, tj. nabor metod za iskanje informacij v velikem številu podatkov, s problematiko v biologiji.

V biologiji pogosto želimo analizirati in identificirati celično sestavo kompleksnih tkiv, ki so lahko sestavljena iz velikega števila različnih celic. Vsaka izmed teh celic je specializirana za opravljanje določenih nalog, kar ji omogočajo aktivacije posameznih genov, ki so odgovorni za potek specializiranih procesov celice. Da ti procesi lahko potekajo, se morajo geni prepisati iz DNA zapisa, ki je v jedru, na RNA zapis celice, ki kasneje sodeluje pri sintezi genskih produktov prepisanih genov npr. proteinov in katalizatorjev. Ker so ti geni povezani samo z določenimi procesi, ki so značilni samo za nekatere specializirane celice, se izrazijo samo na RNA zapisu specifičnih specializiranih celic oz. *celičnih tipov*. Takim genom pravimo *biološki markerji* oz. samo *markerji*, saj je njihov izraz na RNA zapisu koreliran s celičnim tipom celice.

V bioinformatiki je zelo pogost tip podatkov t. i. *sekvenciranje RNA zapisa celic* (ang. *single cell RNA sequencing*), ki vsebujejo podatke o posameznih celicah in njihovih genskih izrazih (ang. *gene expression counts*).

Ključni del analize podatkov o genskih izrazih celice je tudi analiza prisotnih celičnih tipov. Ker je razmerje med geni markerji in njihovimi celičnimi tipi zapleteno, je anotacija celičnih tipov običajno opravljena z nenadzorovanim gručenjem celic (slika 2) in kasnejšo analizo posameznih skupin na podlagi genskih izrazov celičnih markerjev z algoritmi kot je t-SNE [2] in drugimi statističnimi metodami kot je metoda glavnih komponent (PCA) [3]. S tem pristopom lahko dobro razvrstimo in prepoznamo tako



Slika 2: Primer nenadzorovanega gručenja celic z algoritmom t-SNE

Vir: Gonzalez idr., 2018

poznane kot nepoznane celične tipe, vendar je sam proces anotacije celičnih tipov lahko zelo dolg in okoren, poleg tega pa so lahko rezultati zaradi razlik in napak v različnih naborih podatkov zelo nedosledni [4].

Posledično so se na področju bioinformatike in umetne inteligence razvile tako nadzorovane kot nenadzorovane metode za avtomatsko anotacijo celičnih tipov na podlagi podatkov o genskih izrazih. Nadzorovane metode, kot je ACTINN [5], za učenje potrebujejo nabor podatkov, označen s prisotnimi celičnimi tipi, nenadzorovane metode, kot je CellAssign [6], pa potrebujejo bodisi podatke o markerskih genih bodisi prej naučen klasifikator za določeno celično populacijo [4].

Naša raziskava je skušala ugotoviti, koliko lahko pri napovedovanju celičnih tipov pomaga predznanje o markerskih genih in kako to vedenje zakodirati v globoke modele, ki se danes pospešeno uporabljajo v strojnem učenju. Cilj naloge je torej bil razviti globoki model za napovedovanje celičnih tipov, ki bi imel v svoji strukturi vgrajene informacije o vseh celičnih tipih in njihovih markerjih.

Naš model naj bi imel strukturo *samokodirnika* (ang. *autoencoder*) in bi se lahko učil nadzorovano, delno nadzorovano ali nenadzorovano. Za izhodiščni model smo vzeli preprost model, ki za klasifikacijo upošteva samo povprečne izraze markerskih genov posameznega celičnega tipa. Naše hipoteze so bile:

H_1 Struktura nevronske mreže markerjev je ključna – mreža z modificirano strukturo bo delovala slabše.

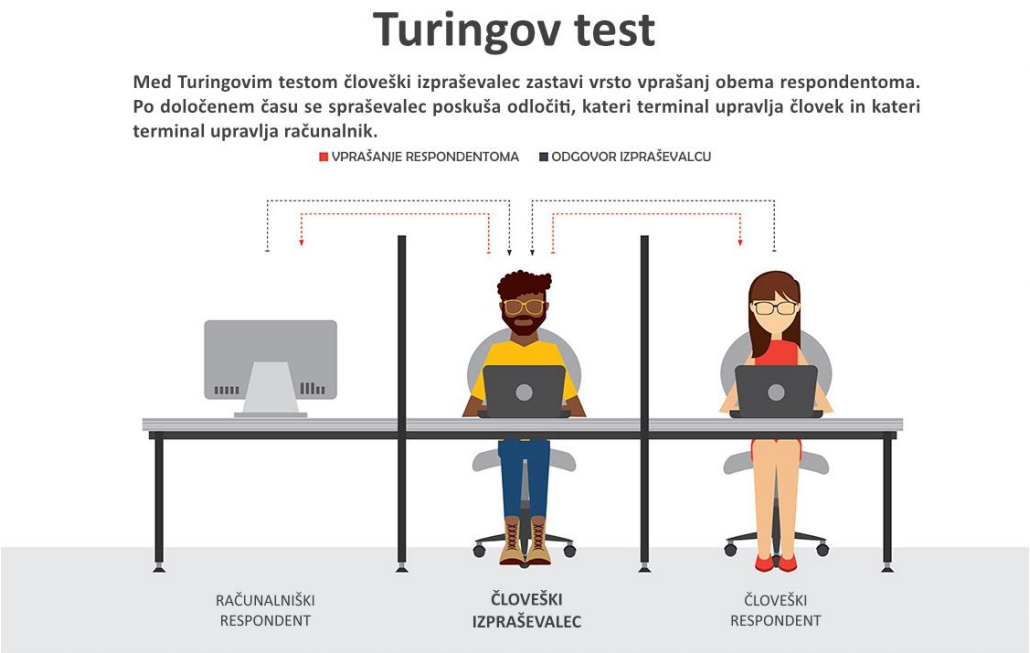
H_2 Nenadzorovani model, ki bo vključeval pravilno mrežo markerjev, bo bolje napovedoval celične tipe kot izhodiščni model.

H_3 Delno nadzorovani model bo dosegel boljše napovedne rezultate kot nenadzorovan model.

2 UMETNA INTELIGENCA

Umetna inteligenca je trenutno eno od najpomembnejših področij v svetu računalništva in informatike. Danes umetno inteligenco lahko opišemo kot poskus avtomatizacije intelektualnih nalog, ki jih običajno opravlja človek (Chollet, 2018, str. 4).

Začetki umetne inteligence sodijo v dobo po 2. svetovni vojni. Izjemno pomemben napredek je članek Alana Turinga, *Computing machinery and intelligence* [7] (1950), v katerem razpravlja, ali so stroji sposobni »mišljenja« in ugotovi, da so. Turing v istem članku opredeli mnoga vprašanja in koncepte, ki so kasneje močno vplivala na razvoj umetne inteligence, in razvije preizkus (slika 3), ki preverja, če je stroj sposoben inteligentnega obnašanja in posnemanja človeka. Danes je ta preizkus znan kot *Turingov test*.



Slika 3: Prikaz Turingovega testa

Vir: TechTarget; poslovenil avtor

V tistem obdobju so strokovnjaki menili, da računalniki lahko dosežejo raven človeške inteligence samo z dovolj velikim naborom eksplicitno napisanih pravil. Temu pristopu pravimo *simbolična umetna inteligenca (*ang. *symbolic artificial intelligence)*. Višek področja se je zgodil v 80. letih s prihodom *ekspertnih sistemov* (ang. *expert systems*), ki naj bi posnemali proces razmišljanja človeških strokovnjakov.

Pred prihodom ekspertnih sistemov, v 60. in zgodnjih 70. letih, in po njih, na začetku 90. let, sta nastopili t. i. *zimi umetne inteligence*. To sta bili obdobji, v katerih so se finančna sredstva in napredek na področju umetne inteligence začasno upočasnili zaradi množičnega navdušenja in posledično razočaranja, saj so ljudje pričakovali, da bo umetna inteligenca v nekaj letih na nivoju visoke inteligence. Leta 1970 je Marvin Minsky, ameriški znanstvenik na področju umetne inteligence, napovedal:

> *Čez tri do osem let bomo imeli stroj s splošno inteligenco povprečnega človeka. V mislih imam stroj, ki bo znal brati Shakespeara, podmazati avto, se iti pisarniške igrice, povedati šalo, se prepirati. V tistem trenutku se bo stroj začel učiti s fantastično hitrostjo. Čez nekaj mesecev bo na genialni ravni, nekaj mesecev po tem pa bodo njegove zmožnosti neprecenljive.*[1]
>
> *– Marvin Minsky v reviji LIFE l. 1970 [8]*

Danes smo dosegli nov višek razvoja umetne inteligence, ki se je začel pred dvajsetimi leti zaradi boljših računalnikov, ki danes lahko računajo in obdelajo podatke hitreje kot kadarkoli prej. Poleg boljših računalnikov nam je zaradi digitalne revolucije na voljo ogromno podatkov (govorimo o t. i. *velikih podatkih*, ang. *big data*) in zato je namesto simbolične UI glavna panoga umetne inteligence t. i. *strojno učenje*.

[1] Prevedel avtor. Izvirno besedilo: In from three to eight years we will have a machine with the general intelligence of an average human being. I mean a machine that will be able to read Shakespeare, grease a car, play office politics, tell a joke, have a fight. At that point the machine will begin to educate itself with fantastic speed. In a few months it will be at genius level and a few months after that its powers will be incalculable.

2.1 Strojno učenje

Pri klasičnem programiranju s kodo in ukazi eksplicitno povemo računalniku, kaj mora narediti, da reši problem, v primeru strojnega učenja pa se računalnik *nauči* rešiti določen problem.

2.1.1 Načini učenja

Učenje v kontekstu strojnega učenja in umetne inteligence pomeni, da računalnik dobi podatke, to so t. i. *učni podatki* oz. *učna množica*, poleg njih pa dobi še pričakovane *izhodne podatke* oz. *anotacije*, ki so »rešitve problema« in so običajno podatki, ki jih mora izdelati človek sam. Vse pričakovane izhodne podatke imenujemo »prave« vrednosti, vse izhodne podatke, ki nam jih da računalnik, imenujemo *napovedi*.

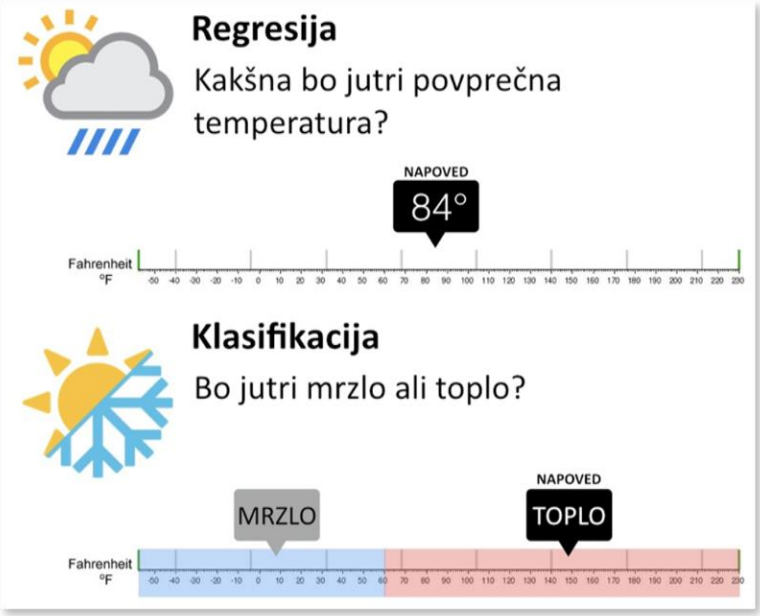
Algoritem za učenje potrebuje tudi *funkcijo izgub*e, ki računalniku pove, kako dobro opravlja svojo nalogo. To funkcijo potrebuje kot povratno informacijo, ki mu omogoča, da spremeni svoje delovanje tako, da izboljša oz. optimizira funkcijo izgube in ima posledično boljše rezultate tako glede učne množice kot glede podatkov, s katerimi se računalnik še nikoli ni srečal. Ko preizkušamo, kako dobro računalnik opravlja svojo nalogo, imenujemo te podatke *testni podatki* oz. *testna množica*.

Opisanemu načinu učenja pravimo *nadzorovano učenje*, saj poskuša računalnik med učenjem aktivno izboljšati svoje rezultate. Primer nadzorovanega učenja je npr. diagnosticiranje na podlagi rentgenskih slik. V tem primeru učno množico sestavljajo rentgenske slike, vsaka slika, imenovana *vzorec* (ang. *sample*), je sestavljena iz več tisoč slikovnih točk (ang. *pixels*), ki jih v strojnem učenju imenujemo *značilke* (ang. *features*) – to

so vsi podatki, ki opisujejo posamezni primer v učni množici. Izhodni podatki so, v tem primeru, diagnoze posamezne rentgenske slike, ki jih postavijo strokovnjaki, funkcija izgube pa je matematična funkcija, ki upošteva odstotek pravilno diagnosticiranih vzorcev v učni množici.

Pri nadzorovanem učenju ločimo dva različna tipa problemov: primere *klasifikacije* in probleme *regresije* (slika 4). Klasifikacija je problem ločevanja posameznih primerov v različne *razrede* (primer z rentgenskimi slikami je primer klasifikacije, posamezne diagnoze oz. bolezni pa so, v tem primeru, razredi). Pri problemih regresije pa model (računalnik) poskuša napovedati *skalar* (število), kot je npr. napovedovanje vrednosti hiše na podlagi podatkov o povprečni plači, izobrazbi ipd. njene okolice.

Glavna razlika med klasifikacijo in regresijo je torej, da v klasifikacijskih problemih računalnik lahko napove samo končno število vrednosti (razredov), v regresijskih problemih pa je število vrednosti neskončno.



Slika 4: Razlika med regresijo in klasifikacijo

Vir: Taylor Fogarty, TowardsDataScience; poslovenil avtor

Poleg nadzorovanega učenja poznamo še *nenadzorovano učenje*, pri katerem model ne upošteva »rezultatov«, ki jih definira človek, ampak običajno poskuša opraviti nalogo gručenja ali zmanjšanja dimenzionalnosti podatkov (npr. iz 1000 dimenzij v 3 ali 2 dimenziji, kateri lahko vizualiziramo na grafu). Z zmanjšanjem dimenzionalnosti lahko bolje razumemo svoje podatke, saj model opravi svojo nalogo tako, da če sta 2 točki blizu v 1000 dimenzijah, bosta blizu tudi v dveh. Posledično dobimo v dveh dimenzijah *skupine* točk (ang. *clusters*), ki so si podobne in zato skupaj, kot npr. na sliki 2 (v primeru na sliki 2 je točka posamezna celica, izvirna dimenzija pa so npr. izrazi 25.000 genov, ki opisujejo celico, torej je izvirni prostor 25.000-dimenzionalen).

Drugi tipi učenja so denimo še *delno nadzorovano učenje*, *samonadzorovano učenje* in *spodbujevalno učenje*.

Delno nadzorovano učenje je mešanica nadzorovanega in nenadzorovanega učenja – nekateri vzorci so anotirani, drugi pa niso, kar velikokrat privede do boljših rezultatov.

Samonadzorovano učenje je podobno nadzorovanemu, le da so anotacije ustvarjene iz učnih podatkov, ne ustvari jih človek. *Samokodirniki*, o katerih več v naslednjih poglavjih, so primer samonadzorovanega učenja.

Spodbujevalno učenje je posebna vrsta učenja, ki se običajno uporablja za učenje igranja video iger, čeprav ima velik potencial za uporabnost na drugih področjih, kot sta robotika in samovozeči avtomobili. Pri spodbujevalnem učenju dobi model podatke o svoji okolici in se nauči igrati igro tako, da dobi največje število točk (npr. hitreje kot reši nivo igre, več točk dobi).

2.1.2 Modeli strojnega učenja

Za to, da se računalnik lahko uči, potrebuje algoritem oz. *model*. Na področju strojnega učenja obstaja več vrst algoritmov oz. modelov.

Primeri modelov, ki izhajajo izpred časov strojnega učenja, kot ga poznamo danes, so v*erjetnostni modeli*, ki uporabljajo načela iz statistike in so ena od najstarejših oblik učenja in do danes še vedno zelo uporabljani. Najbolj znana verjetnostna modela sta klasifikatorja *naivni Bayes*, ki uporablja Bayesov izrek[2], in *logistična regresija*, ki uporablja logaritmično funkcijo, da loči dva razreda v prostoru.

Drugi tip modelov, ki je bolj zasnovan v matematiki kot v računalništvu, so t. i. *jedrne metode* (ang. *kernel methods*), izmed katerih je najbolj znana metoda podpornih vektorjev (ang. *support vector machine* oz. *SVM*), ki se uporablja še danes. SVM je klasifikacijska metoda, ki išče hiperravnino (tj. ravnina v poljubno dimenzionalnem prostoru), ki najbolje loči dva razreda. Formulacija SVM-a, ki se uporablja danes, je bila objavljena leta 1995 [9], začetki metode pa izvirajo že iz leta 1963.

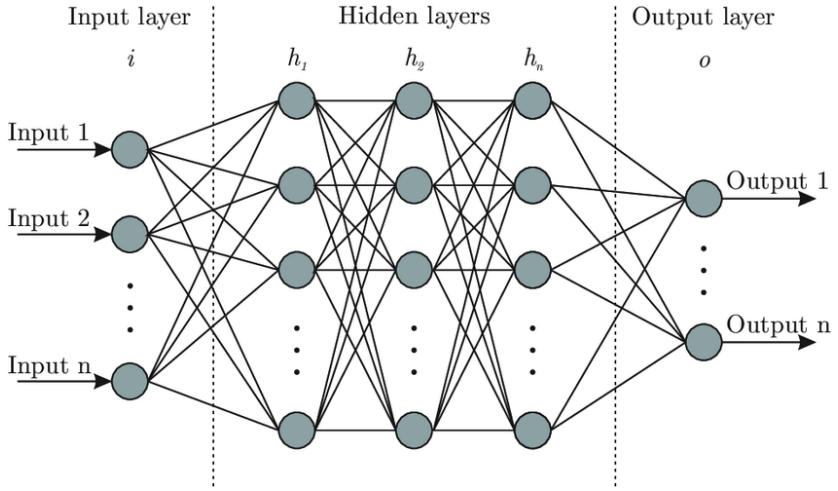
Danes je najbolj popularna vrsta modela t. i. *umetna nevronska mreža*, ki je trenutno ena od ključnih metod na področju strojnega učenja.

[2] Bayesov izrek je izračun verjetnosti dogodka A, če se je zgodil dogodek B, definiran s predpisom $P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$, če velja $P(B) \neq 0$

2.2 Nevronske mreže

2.2.1 Struktura nevronskih mrež

Nevronska mreža, bolj specifično *usmerjena nevronska mreža*, je eden od ključnih modelov v strojnem učenju. Nevronske mreže so se rodile iz ideje, da naj bi se nevroni v človeških možganih povezovali v velike nevronske mreže. Nevronske mreže gradijo t. i. *nevroni*, ki so povezani v *sloje* (slika 5). Vsak sloj je sestavljen iz določenega števila nevronov, ki so vsi povezani s slojem pred in za njimi, razen prvega (vhodnega) in zadnjega (izhodnega) sloja, ki sta povezana s samo enim slojem. Število nevronov vhodnega sloja je odvisno od števila značilk, ki jih podamo mreži (vsako značilko predstavlja en nevron). Pri klasifikaciji vsak nevron izhodnega sloja običajno predstavlja en razred, pri regresiji pa vsak nevron predstavlja en skalar, ki ga hočemo napovedati. Vse sloje razen vhodnega in izhodnega imenujemo *skriti sloji*.



Slika 5: Primer nevronske mreže z vhodnim slojem (input layer), tremi skritimi sloji (hidden layers) in izhodnim slojem (output layer)

Vir: Bre idr., 2018

Taki strukturi pravimo *polno povezana nevronska mreža*, obstajajo pa tudi *delno povezane nevronske mreže*, pri katerih ni nujno, da je vsak nevron povezan z obema slojema.

Vrednost *j-tega* nevrona *l-tega* sloja definiramo s predpisom

$$a_j^l = \phi\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), \tag{1}$$

kjer je a_j^l vrednost oz. *aktivacija* nevrona, w_{jk}^l je t. i. *utež* povezave med j-tim nevronom l-tega sloja in k-tim nevronom (l – 1)-tega sloja, b_j^l pa je t. i. *pristranskost* nevrona. Vsak nevron ima svoje vrednosti $w\ in\ b$, ki jih med učenjem spreminjamo, da spreminjamo izhodne vrednosti nevronske mreže. Vse vrednosti $w\ in\ b$ imenujemo *parametri nevronske mreže* (označeni z θ). ϕ v tem primeru predstavlja t. i. *aktivacijsko funkcijo*, ki daje mreži nelinearnost. Aktivacijska funkcija mora biti odvodljiva, saj je to pogoj za delovanje algoritma, s katerim se nevronska mreža uči. Pogosto uporabljeni aktivacijski funkciji sta *sigmoidna funkcija*, ki slika iz realnih števil v interval $(0, 1)$ in je definirana s predpisom

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2}$$

in *popravljena linearna enota* (ang. *rectified linear unit* oz. *ReLU*), ki je definirana s predpisom

$$R(x) = max(0, x), \tag{3}$$

Enačbo (*(1)*) lahko posplošimo v kontekstu posameznega skritega sloja h^l s predpisom

$$h^l = \phi(W^T h^{l-1} + b), \tag{4}$$

kjer sta W in b matrika uteži in vektor pristranskosti sloja l, h^l pa je vektor aktivacij nevronov sloja l.

2.2.2 Učenje nevronskih mrež

Problem, ki ga rešujemo z nevronskimi mrežami, si lahko predstavljamo kot funkcijo vhodnih podatkov, f^*. Če je to problem klasifikacije, funkcija $f^*(x) = y$ slika vhodne podatke x v razred y. Potem nevronska mreža, ki rešuje problem, definira funkcijo $f(x;\theta) = \hat{y}$ in išče vrednosti takih parametrov θ, da velja $f(x;\theta) \approx f^*(x)$.

Ko torej govorimo o učenju nevronske mreže, govorimo o iskanju parametrov θ, za kar potrebujemo funkcijo izgube, ki je odvisna od vrste problema. Pri problemih klasifikacije je zelo pogosta funkcija izgube *prečna entropija* (ang. *cross entropy*), ki je definirana s predpisom

$$D(\hat{y}, y) = -\sum_{i}^{N} y_i \, ln\hat{y}_i, \tag{5}$$

kjer je N število vzorcev, y vektor pravih razredov[3], $\hat{y}$ pa je vektor verjetnostnih porazdelitev za vsak razred.

Da dobimo verjetnostne porazdelitve, običajno v zadnjem sloju nevronske mreže kot aktivacijsko funkcijo uporabljamo funkcijo *softmax*, ki slika aktivacijo vsakega nevrona (razreda) iz realnih števil v interval $(0, 1)$, ki predstavlja verjetnost tega razreda, in je definirana s predpisom

$$\phi(a_i) = \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}, \qquad \text{za } i = 1, \dots, N. \tag{6}$$

[3] Vektor pravilnih razredov je običajno vektor vektorjev ničel z eno enko na mestu pravilnega razreda.

Pri problemih regresije pa je najbolj pogosto uporabljena funkcija izgube *povprečna kvadratna napaka* (ang. *mean square error* oz. *MSE*), ki je definirana s predpisom

$$MSE(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}(y - \hat{y})^2. \tag{7}$$

Funkcija izgube nam pove, kako dobro model opravlja svojo nalogo z določenimi parametri θ. Npr., če je funkcija izgube modela MSE in če model slabo opravlja svojo nalogo, torej bo povprečen $\hat{y}$ veliko manjši ali veliko večji od povprečnega y, bo vrednost MSE-ja veliko večja od nič. Če pa bo model dobro opravljal svojo nalogo, bo vrednost MSE-ja blizu nič, torej iščemo take parametre θ, da bo vrednost MSE-ja čim bližja nič, da bo posledično tudi veljalo $f(x;\theta) \approx f^*(x)$. Funkcijo izgube $\mathcal{L}$ lahko napišemo kot funkcijo parametrov θ, npr. na primeru MSE-ja

$$\begin{aligned} MSE\big(f(x;\theta), f^*(x)\big) &= \frac{1}{N}\sum_{i=1}^{N}\big(f^*(x) - f(x;\theta)\big)^2 \\ MSE(\theta) &= \frac{1}{N}\sum_{i=1}^{N}\big(y - f(x;\theta)\big)^2, \end{aligned} \tag{8}$$

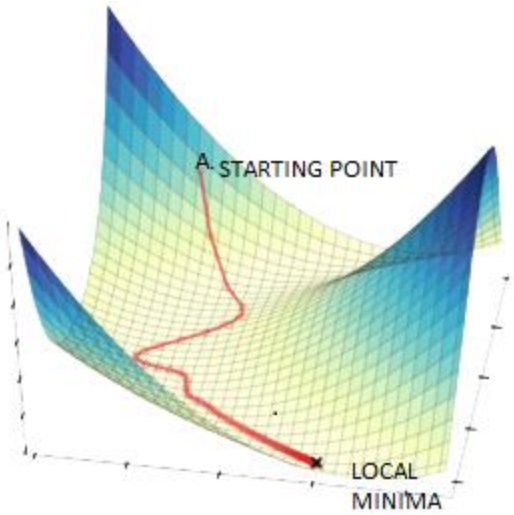
saj lahko učne podatke x in prave vrednosti y tretiramo kot konstante.

Z drugimi besedami: s parametri θ iščemo globalni minimum (najmanjšo vrednost) poljubne funkcije izgube $\mathcal{L}$, kar zapišemo s predpisom

$$f(x;\theta) = arg\,\underset{\theta}{min}\,\mathcal{L}(\theta).$$

Učenje nevronske mreže, torej iskanje parametrov θ, poteka z algoritmoma, ki sta imenovana *spust po gradientu* in *vzvratno razširjenje* [10].

Spust po gradientu (tudi spust po gradientu; slika 6) je algoritem za iskanje lokalnih minimumov poljubne odvodljive funkcije. Delovanje spusta po gradientu si lahko intuitivno predstavljamo s preprosto alegorijo: predstavljamo si, da hodimo v hribih in se želimo spustiti v dolino, ki predstavlja naš minimum, ampak je okoli nas zelo megleno in vidimo samo nekaj metrov okoli sebe. Logična rešitev je, da pogledamo okoli sebe in začnemo hoditi v tisto smer, kjer se relief spušča, dokler ne pridemo do točke, kjer ne moremo iti več dol. Točka, kjer se ustavimo ni nujno najnižja točka (globalni minimum), je pa najnižja točka okoli nas (lokalni minimum).



Slika 6: Prikaz spusta po gradientu

Vir: Rashmi Jain, Hackerearth

Podobno deluje tudi spust po gradientu pri nevronskih mrežah – v določeni točki funkcije izgube $\mathcal{L}(\theta)$ lahko izračunamo njen gradient, ki nam pove v katero »smer« se moramo premakniti, da pridemo do lokalnega minimuma, saj lahko od parametrov θ odštejemo njihove odvode (enačba 9), da se bližamo minimumu, kjer so njihovi odvodi enaki 0. Spust po gradientu je iterativni algoritem, kar pomeni, da njegov postopek ponavljamo, dokler ne pridemo do rezultata (minimuma), s katerim smo zadovoljni.

Gradient funkcije v umetni inteligenci pomnožimo še z majhnim številom α (t. i. hitrost učenja oz. ang. *learning rate*), saj tako preprečujemo, da bi algoritem preveč »skakal« po funkciji in zato dobil slabše rezultate. α je običajno konstanta, obstajajo pa tudi algoritmi, ki med učenjem spreminjajo njeno vrednost [11].

$$\theta' \coloneqq \theta - \alpha \nabla L(\theta) \tag{9}$$

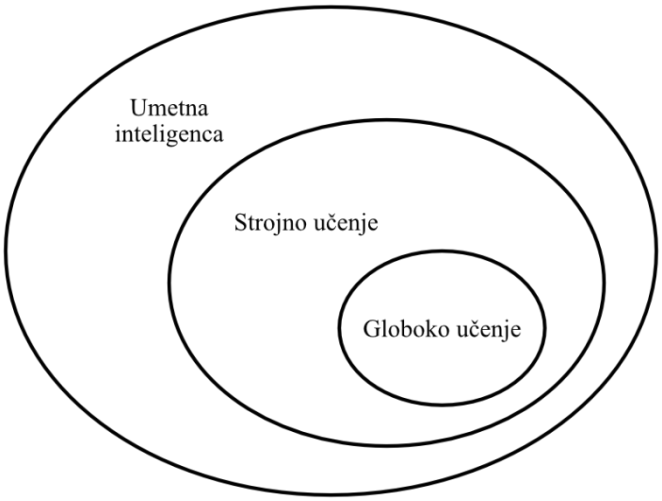
Za računanje parametrov mreže je odgovoren algoritem, imenovan vzvratno razširjanje, ki izračuna odvode posameznih parametrov tako, da uporablja verižno pravilo odvajanja, saj je nevronska mreža sestavljena funkcija.

Danes se v praksi uporablja različica spusta po gradientu, imenovana *stohastični spust po gradientu*. Spust po gradientu predvideva, da za izračun gradienta funkcije izgube uporabimo celotno učno množico, kar je lahko računsko zelo neoptimalno, saj ima naša učna množica lahko več milijonov vzorcev, med učenjem pa moramo gradient izračunati tudi do več tisočkrat. Stohastični spust po gradientu ta problem reši tako, da vsakič ko mora izračunati gradient, vzame določeno število naključnih vzorcev, imenovano *paket* (ang. *batch*), ki običajno vsebuje 128, 256, 512 itd. vzorcev, in z njim izračuna približek gradienta.

Na začetku učenja, začnemo s poljubnimi parametri θ, potem pa izračunamo parametre z algoritmoma stohastični spust po gradientu in vzvratno razširjanje tako, da izračunamo gradient funkcije izgube na celotni učni množici in sproti spremljamo uspešnost nevronske mreže (npr. vrednost funkcije izgube, klasifikacijska točnost) računamo nove vrednosti parametrov nevronske mreže. Vsakič ko izračunamo gradient na celotni učni množici, temu pravimo ena *epoha* (ang. *epoch*). Nevronsko mrežo učimo toliko epoh, dokler nismo zadovoljni z rezultati.

Med učenjem spremljamo uspešnost nevronske mreže tako na učnih podatkih kot na drugih podatkih, ki jih imenujemo validacijska množica, saj se nevronska mreža lahko med učenjem začne prekomerno prilegati (ang. *overfitting*) učni množici in ima zato slabše rezultate na podatkih, ki niso v učni množici. S prekomernim prileganjem se ukvarja nabor metod, ki se imenujejo *regularizacijske metode*, ki spodbujajo boljšo generalizacijo med učenjem nevronske mreže.

2.3 Globoko učenje



Slika 7: Umetna inteligenca, strojno učenje in globoko učenje

Globoko učenje je del strojnega učenja (slika 7), ki temelji na globokih nevronskih mrežah (to so vse nevronske mreže, ki imajo vsaj en skriti sloj).

Globoke nevronske mreže naj bi bile sposobne t. i. *učenja predstavitev*, kar npr. pomeni, da se mreža pri nalogi klasifikacije slik ne uči samo npr. prepoznavanja črt in mej med podobami na sliki, ampak da se nauči prepoznavati podobe kot npr. »uho«, »nos«, »obraz«, »človek«, »pes« …

Globoke nevronske mreže imajo običajno posebne strukture, npr. *konvolucijske nevronske mreže*, ki uporabljajo posebne sloje, imenovane konvolucijski sloji. Konvolucijske nevronske mreže se običajno uporabljajo na področju, imenovanem *računalniški vid*, ki se ukvarja z računalniškim razumevanjem videov in slik ter želi avtomatizirati naloge, ki jih običajno opravlja vidni sistem človeka. Področje računalniškega vida ima zadnje čase veliko vlogo na področjih kot so samovozeči avtomobili, računalniška animacija, robotika, varnostna tehnologija (npr. prepoznavanje obrazov) idr.

2.3.1 Samokodirniki

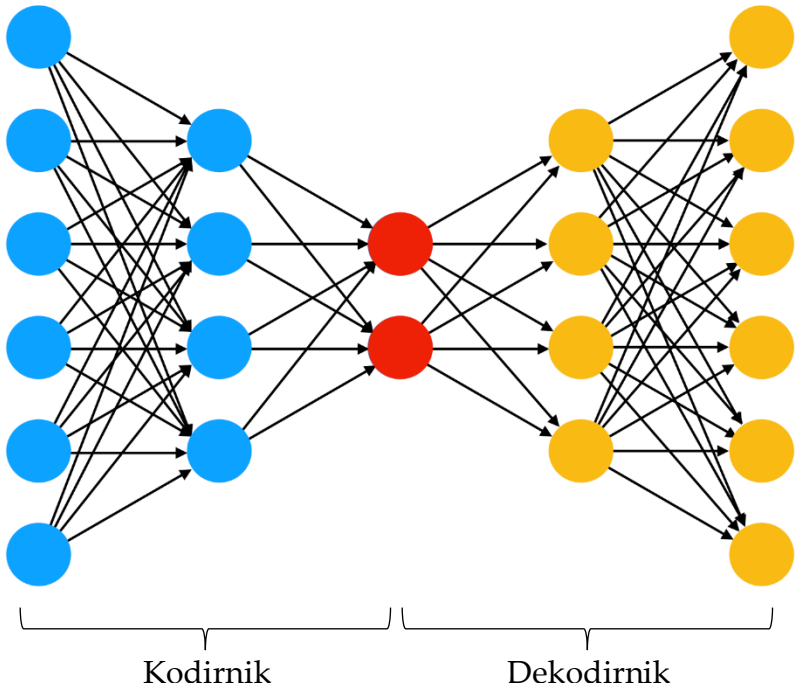
Samokodiriniki so poseben tip globokih modelov (slika 8) in so prav tako sposobni učenja podob. Ideja je, da se prvi del modela (*kodirnik*) nauči *vložiti* podatke v prostor, imenovan *prikriti prostor* (ang. *latent space*), drugi del modela (*dekodirnik*) pa se nauči podatke preslikati iz prikritega prostora v prvotni prostor in tako naj bi se posledično model naučil »podob« vzorcev.

Kodirnik in dekodirnik lahko definiramo kot preslikavi ε in δ, za kateri velja:

$$\varepsilon : \mathrm{F} \rightarrow \Gamma$$

$$\delta : \Gamma \rightarrow \mathrm{F},$$

kjer je F prvotni prostor ($\mathrm{F} = \mathbb{R}^d$), Γ pa predstavlja prikriti prostor ($\Gamma = \mathbb{R}^b$). Prikriti prostor ima praviloma manjšo dimenzijo kot prvotni prostor vzorcev ($b > d$), saj



Slika 8: Primer strukture samokodirnika

Vir: CompTree

samokodirnik potrebuje »oviro«, da se lahko nauči podob, zato del modela z najmanjšo dimenzijo imenujemo *ozko grlo*.

Preprosti samokodirniki se učijo tako, da zmanjšajo razliko med prvotnimi podatki, ki jih dobi kodirnik, in izhodnimi podatki, ki so t. i. *rekonstrukcije* prvotnih podatkov, zato se za funkcijo izgube običajno uporablja MSE (enačba 7).

Zaradi njihove dobre sposobnosti učenja podob se samokodirniki in njihove različice uporabljajo na mnogih področjih. Preprosti samokodirniki se lahko uporabljajo za zmanjšanje dimenzionalnosti podatkov s podobnimi ali boljšimi rezultati kot drugi, bolj pogosto uporabljeni algoritmi kot sta t-SNE in PCA [12].

Prav tako se samokodirniki uporabljajo za *imputacijo* (obnovitev) podatkov, kot so slike slabe kvalitete in tudi podatki o genskih zapisih [13].

Variacijski samokodirniki [14], ki poleg rekonstrukcije za funkcijo izgube upoštevajo še Kullback–Leiblerjevo divergenco, se uporabljajo za generacijo novih vzorcev iz prikritega prostora, npr. na sliki 8, kjer je variacijski samokodirnik generiral popolnoma nove obraze.

Slika 9: Generacija novih obrazov z variacijskim samokodirnikom

Vir: Irhum Shafkat, TowardsDataScience

3 IDENTIFIKACIJA CELIČNIH TIPOV S SAMOKODIRNIKOM

V nalogi smo želeli razviti globoki model, ki bi v svoji strukturi imel vgrajene podatke o genskih markerjih in bi se lahko z nenadzorovanim učenjem naučil prepoznavati celične tipe na podlagi izrazov celičnih markerjev.

Za izhodiščni model smo vzeli preprost model, ki upošteva le povprečne izraze genskih markerjev posameznega celičnega tipa. Tak model nima dobre klasifikacijske točnosti celičnih tipov, saj so razmerja med celičnimi tipi in njihovimi markerji bolj zapletena kot tak model domneva. Npr. nekateri celični tipi imajo genske markerje, ki so skupni več celičnim tipom, nekateri celični tipi imajo pet genskih markerjev, drugi pa jih imajo lahko 200, poleg tega pa ima lahko majhen izraz enega gena veliko večjo korelacijo z njegovim celičnim tipom kot pa veliko večji izraz drugega genskega markerja istega celičnega tipa.

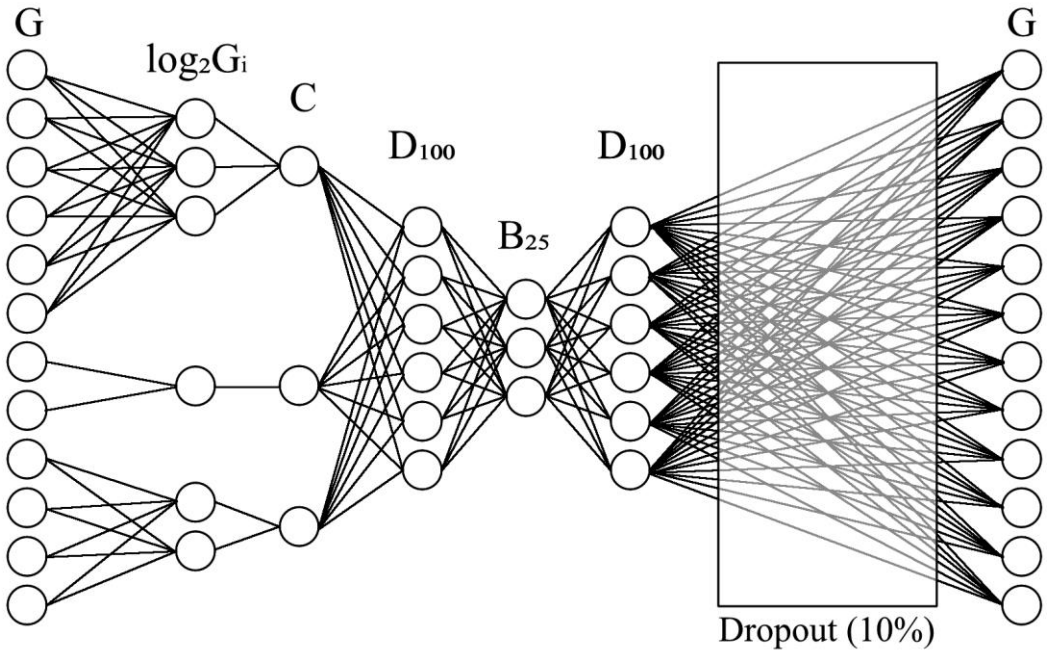
Za osnovni model smo si izbrali samokodirnik zaradi njegove sposobnosti učenja podob – ideja je, da so podobe, ki se jih samokodirnik nauči pri podatkih o genskih izrazih, sami celični tipi. Poleg tega pa lahko v preprosto strukturo samokodirnika intuitivno vkodiramo razmerja med genskimi markerji in njihovimi pripadajočimi celičnimi tipi, hkrati pa s to metodo dobimo tudi razlago za klasifikacijo celičnih tipov (naš model ni črna skrinjica).

3.1 Metoda

3.1.1 Struktura modela

Za osnovno strukturo smo vzeli samokodirnik, ki ima na vhodu in izhodu izraze genskih markerjev. Število vozlišč na vhodu in izhodu je odvisno od števila genskih markerjev v naboru podatkov in je izračunano za vsak eksperiment posebej.

Informacije o genskih markerjih in njihovih celičnih tipih smo v strukturo (slika 10) vkodrirali tako, da smo dodali skriti sloj, kjer je vsako vozlišče predstavljalo en celični tip. Vsako vozlišče smo nato povezali z njegovimi genskimi markerji preko vmesnega, delno povezanega, skritega sloja. Vmesni skriti sloj je imel posamezne skupine vozlišč za vsak celični tip, število vozlišč enega celičnega tipa pa je bilo enako $\log_2 G_i$, kjer G_i predstavlja število markerjev celičnega tipa i. Vmesni sloj uporabljamo z namenom, da se model nauči globokih razmerij med posameznimi geni in njihovimi celičnimi tipi.



Slika 10: Struktura mreže markerjev

Sloju celičnih tipov sledi polno povezan sloj s 100 nevroni, njemu pa drug polno povezan sloj s 25 nevroni, ki je tudi najožji sloj mreže (ozko grlo). Temu sloju sledi polno povezan sloj s 100 nevroni, ki je povezan z izhodnim slojem, ki predstavlja rekonstrukcije genskih izrazov. Na zadnjem sloju za regularizacijsko metodo uporabljamo 10% izpust povezav (ang. *dropout regularization*) [15], ki mreži pomaga posploševati svoje predstavitve.

Delno povezane sloje smo dosegli tako, da smo polno povezanemu sloju definirali binarno matriko, ki je delovala kot filter povezav sloja.

Za aktivacijsko funkcijo smo na vseh slojih modela uporabljali *ReLU*, za funkcijo izgube pa smo pri vseh modelih uporabljali funkcijo MSE, ki je primerjala vhodni sloj genov z njihovo rekonstrukcijo na izhodu.

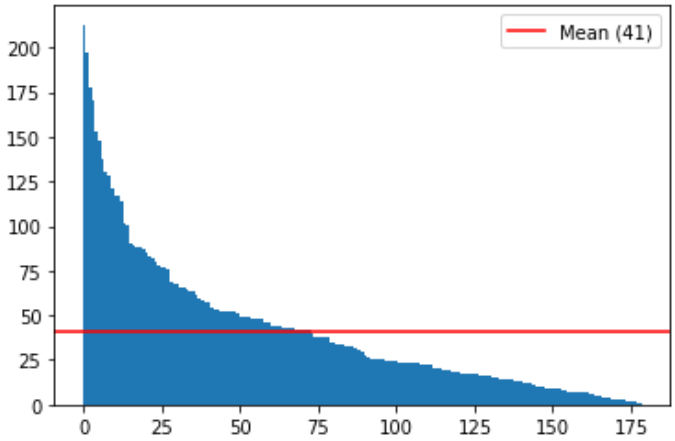
Edina razlika v delovanju nenadzorovanega, delno nadzorovanega in nadzorovanega modela je bila, da sta delno nadzorovani in nadzorovani model, poleg MSE-ja, upoštevala tudi prečno entropijo na sloju, ki je predstavljal aktivacije celičnih tipov. Delno nadzorovani model je prečno entropijo upošteval le na celicah, ki so imele označen celični tip, nadzorovan model pa se je učil samo z označenimi celicami in je zato prečno entropijo upošteval na celi učni množici.

Za optimizator stohastičnega spusta po gradientu smo uporabljali optimizator Adam [16].

3.1.2 Priprava podatkov

Pri izbiranju markerskih genov smo upoštevali samo markerje, ki so bili skupni vsem naborom podatkov, uporabljenim v eksperimentu, ostale gene pa smo odstranili. Odstranili smo tudi vse označene celice, za katere nismo imeli nobenega gena markerja v naboru podatkov. Podatke o genskih markerjih smo dobili iz nabora podatkov PanglaoDB [24], ki vsebuje podatke o genskih markerjih človeških in mišjih celičnih tipov.

Končni nabor markerskih genov je bil torej odvisen od naborov podatkov pri posameznem eksperimentu, povprečno pa smo uporabljali ~4000 genov, ki so bili markerji ~175 celičnih tipov, in ~7300 povezav med geni in njihovimi celičnimi tipi. Na sliki 11 je histogram, ki prikazuje število markerskih genov, ki jih je imel posamezen celični tip pri eksperimentu z nenadzorovanim modelom, pri katerem je povprečni celični tip imel okoli 41 markerjev.



Slika 11: Histogram markerskih genov na celični tip pri poskusu z nenadzorovanim modelom

Nato smo genske ekspresije posameznih genov normalizirali na število ekspresij na milijon celic oz. *CPM* (ang. *counts per million*). CPM posameznega gena *X*, ki se pojavi v *N* celicah, je definiran s predpisom:

$$CPM(X) = \frac{X}{N} 10^6 \tag{10}$$

Genske izraze, normalizirane s CPM normalizacijo, smo nato pretvorili še z naravnim logaritmom:

$$X' = \log_{\mathrm{e}}(X + 1), \tag{11}$$

saj se nevronske mreže hitreje in bolje učijo z manjšimi števili.

V eksperimentih smo uporabljali nabore podatkov sekvenciranja mišjega RNA zapisa (preglednica 1). Nabor podatkov Zheng smo uporabljali pri eksperimentih nenadzorovanega in delno nadzorovanega učenja, pri delno nadzorovanih eksperimentih smo za označene celice uporabljali celice iz nabora Tabula muris, saj vsebujejo celice iz več kot 20 mišjih organov. Nabora Chen in Hrvatin pa smo uporabljali pri nadzorovanem učenju in kot testne množice pri vseh eksperimentih.

Preglednica 1 *Pregled uporabljenih naborov podatkov*

Nabor podatkov	Št. celic	Št. genov	Opis	Vir
Zheng	1.306.127	27.932	Mišji PBMC[4]	[25]
Hrvatin	48.266	25.186	Mišji primarni vizualni korteks	[26]
Tabula muris	44.949	23.433	Atlas mišjega telesa	[27]
Chen	14.437	23.284	Mišji hipotalamus	[28]

[4] Periferne krvne enojedrne celice (ang. peripheral blood mononuclear cells)

3.2 Implementacija modela in eksperimentov

Za implementacijo modela in izvedbo eksperimentov smo izdelali manjšo knjižnico, *saly* (Priloga A), v programskem jeziku Python. Model smo zgradili s knjižnicama Keras [17] in TensorFlow [18]. Za nalaganje in obdelovanje podatkov smo uporabili knjižnice NumPy [19], Pandas [20], SciPy [21], ScanPy in AnnData [22], za vizualizacijo podatkov pa smo uporabili knjižnico MatPlotLib [23].

Knjižnico smo razdelili na dva dela: *deep* in *backend. Deep* deluje kot ospredje knjižnice in je del knjižnice, kjer se nahajajo vse funkcije, potrebne za izvedbo eksperimentov. *V backend* delu pa se nahajajo vse manjše funkcije, ki sestavljajo in se večkrat pojavijo v *deep* delu knjižnice. *Backend* del sestavljajo štiri datoteke, ki organizirajo funkcije knjižnice po njihovem namenu (*analysis.py*, *data.py*, *model.py* in *preprocessing.py*). Vse štiri datoteke v *backend* delu imajo tudi svoje ekvivalente z istimi imeni tudi v *deep* delu. Namen take strukture je, da je velika večina funkcij, ki se uporabljajo v datoteki v *deep* delu, definirana v *backend* delu v datoteki z istim imenom, kar da knjižnici mnogo boljšo orientacijo.

3.2.1 Razpoložljivost podatkov

Vsa koda in eksperimenti so razpoložljivi na avtorjevem GitHub profilu (*https://github.com/LongarMD/MarkerNetwork*).

4 REZULTATI

Za oceno rezultatov smo uporabljali statistično vrednost F_1, ki je definirana kot harmonična sredina točnosti in priklica s predpisom:

$$F_1 = 2\,\frac{točnost * priklic}{točnost + priklic}$$

točnost in priklic pa sta definirana s predpisoma:

$$točnost = \frac{PP}{PP + NP} \text{ in}$$

$$priklic = \frac{PP}{TP + NN},$$

kjer je PP število pravilno pozitivnih, NP število napačno pozitivnih, NN pa število napačno negativnih vzorcev. F_1 ima zalogo vrednosti na intervalu [0, 1], kjer vrednost 0 pomeni, da model ni pravilno klasificiral niti enega vzorca, vrednost 1 pa pomeni, da je model pravilno klasificiral vse vzorce.

Vrednost F_1 smo uporabljali, ker z upoštevanjem tako točnosti kot priklica lahko bolje predstavlja uspešnost modela kot če bi uporabljali le točnost, saj je lahko ta zavajajoča pri podatkih z neenakomernim številom vzorcev v posameznih razredih.

H_1 – Struktura nevronske mreže markerjev je ključna – mreža z modificirano strukturo bo delovala slabše

Za učno množico smo uporabljali nabor podatkov Zheng in pred učenjem mreže pomešali njegove gene. Med učenjem smo spremljali klasifikacijsko točnost podatkov Chen, ki se med učenjem ni veliko spreminjala in je krožila okoli 10%. Med učenjem smo spremljali tudi vrednost MSE-ja, ki se med učenjem ni veliko spremenil, kar kaže k temu, da se nevronska med učenjem ni znala naučiti razmerij med markerji in njihovimi celičnimi tipi. Najboljše F_1 vrednosti na testni množici (preglednica 2) so imeli nevroni in astrociti, saj je mreža velik del celic klasificirala za nevrone oz. astrocite, posledično jih je mreža tudi relativno dobro klasificirala.

Mreža se očitno ni mogla naučiti razmerij med geni in celičnimi tipi, zato smo hipotezo lahko **potrdili**.

Preglednica 2 *F_1 vrednosti podatkov eksperimenta z zamenjanimi geni.*

Celični tip	**Chen**	**Hrvatin**
Nevroni	0,49	0,45
OPC celice	0,00	0,00
Oligodendrociti	0,00	0,00
Mikroglije	0,00	0,00
Makrofage	0,00	0,00
Astrociti	0,02	0,01
Endotelialne celice		0,11
Periciti		0,00

H_2 – Nenadzorovani model, ki bo vključeval pravilno mrežo markerjev, bo bolje napovedoval celične tipe kot izhodiščni model

Za učno množico smo uporabljali nabor podatkov Zheng. Nenadzorovano mrežo smo učili 6 epoh. Med učenjem smo spremljali vrednost klasifikacijske točnosti na podatkih Chen in vrednost MSE-ja na učnih podatkih. Z naraščanjem klasifikacijske točnosti je padala tudi vrednost MSE-ja, kar kaže na to, da se je mreža naučila podob posameznih celičnih tipov prek skritega sloja, ki je predstavljal celične tipe, in je zato imela tudi boljšo klasifikacijsko točnost.

F_1 vrednosti na učni množici (preglednica 3) so bile bodisi mnogo boljše kot vrednosti izhodiščnega modela (npr. nevroni, oligodendrociti, periciti), ki je upoštevala le povprečne ekspresije markerjev, bodisi so bile primerljive z izhodiščnim modelom (mikroglije, makrofage). Model je najslabše klasificiral makrofage, s katerimi je imel probleme tudi izhodiščni model, vendar je naš model imel vseeno boljše rezultate. Nenadzorovani model je na testnih podatkih Chen in Hrvatin imel boljšo klasifikacijsko točnost pri skoraj vseh razredih, zato smo to hipotezo lahko **potrdili**.

	Nenad. model		**Izhodiščni model**	
Celični tip	Chen	Hrvatin	Chen	Hrvatin
Nevroni	**0,95**	**1,00**	0,59	0,19
OPC celice	0,95	**0,85**	**0,97**	0,84
Oligodendrociti	**0,98**	**0,94**	0,87	0,78
Mikroglije	**0,88**	0,84	0,81	**0,93**
Makrofage	**0,34**	**0,24**	0,33	0,03
Astrociti	**0,94**	**1,00**	0,73	0,86
Endotelialne celice		**0,91**		0,67
Periciti		**0,99**		0,03

Preglednica 3 *F_1 vrednosti podatkov eksperimenta z nenadzorovanim modelom*

H_3 – Delno nadzorovani model bo dosegel boljše napovedne rezultate kot nenadzorovan model

Pri poskusu smo uporabili nabor podatkov Zheng za učenje nenadzorovanega modela, za učenje delno nadzorovanega modela pa smo uporabljali nabor podatkov Zheng za nenadzorovan del učenja in nabor podatkov Tabula muris za nadzorovani del učenja. Nenadzorovani del modela smo učili 6 epoh, nadzorovani del pa 10.

Delno nadzorovani model je imel povprečno boljše F_1 vrednosti kot izhodiščni in nenadzorovani model. Najbolj opazne spremembe F_1 vrednosti so imele mikroglije in makrofage, ki jih je nenadzorovani model zelo slabo prepoznaval, čeprav je izhodiščni model imel na podatkih Chen pri makrofagah vseeno boljše rezultate.

Delno nadzorovan model je uporabil znanje, ki ga je pridobil nenadzorovani model, za to, da je izboljšal tako klasifikacijsko točnost kot MSE na testni množici, zato smo to hipotezo lahko **potrdili**.

	Delno nad. model		**Nenad. model**		**Izhodiščni model**	
Celični tip	Chen	Hrvatin	Chen	Hrvatin	Chen	Hrvatin
Nevroni	**0,98**	0,99	0,94	**1,00**	0,59	0,19
OPC celice	**0,98**	0,91	0,91	**0,97**	0,97	0,84
Oligodendrociti	0,98	0,97	0,97	**0,98**	0,87	0,78
Mikroglije	**0,93**	**0,97**	0,70	0,01	0,81	0,93
Makrofage	0,15	**0,43**	0,00	0,02	**0,33**	0,03
Astrociti	**0,96**	**1,00**	0,90	0,99	0,73	0,86
Endotelialne celice		**1,00**		0,98		0,67
Periciti		**1,00**		0,93		0,03

Preglednica 4 *F_1 vrednosti podatkov eksperimenta z delno nadzorovanim modelom*

5 ZAKLJUČEK

Zaradi vse večjih naborov podatkov, ki se pojavljajo na področjih biologije in bioinformatike, se je pojavila potreba po avtomatični analizi in klasifikaciji celičnih tipov, prisotnih v kompleksih tkivih. V nalogi smo zato razvili metodo za avtomatično klasifikacijo celičnih tipov na podlagi podatkov o genskih izrazih. Naša metoda je sposobna prepoznavanja celičnih tipov z uporabo nenadzorovanega učenja na velikih podatkih in je v klasifikacijski točnosti primerljiva z drugimi sodobnimi metodami [4].

V nalogi smo na nekaj primerih pokazali, da so informacije o markerskih genih in njihovih celičnih tipih, ki jih vkodiramo v strukturo, ključne za dobre rezultate modela. Pokazali smo tudi, da se naš model lahko nauči kompleksnih povezav med markerskimi geni in njihovimi celičnimi tipi tako z nadzorovanim učenjem kot z nenadzorovanim učenjem in delno nadzorovanim učenjem.

Pomanjkljivost naše metode je, da je njena uspešnost v klasifikaciji odvisna od prisotnosti celičnih tipov v učnih podatkih. Ta pomanjkljivost in klasifikacijska točnost metode na testnih potrjuje nedavna opažanja, da so globoke nevronske mreže mogoče prezapletene za problem klasifikacije celičnih tipov na podlagi podatkov o genskih izrazih [4], [29], katerega verjetnostni modeli in, na splošno, klasične metode strojnega učenja v povprečju boljše rešujejo.

V nadaljnjem raziskovanju bi lahko model poskušali izboljšati tako, da bi bil zmožen prepoznati celični tip, ki ga med učenjem ni srečal, kar bi delno rešilo njegovo glavno pomanjkljivost. To bi lahko naredili tako, da bi v sloj celičnih tipov dodali še en nevron, ki bi predstavljal »neznan« celični tip, kot to naredijo sodobne metode, ki se ukvarjajo s klasifikacijo celičnih tipov s pomočjo nevronskih mrež [5].

6 VIRI

[1] P. Rajpurkar *in sod.*, „CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning“, *ArXiv171105225 Cs Stat*, dec. 2017.

[2] L. van der Maaten in G. Hinton, „Visualizing data using t-SNE“, 2008.

[3] I. T. Jolliffe in J. Cadima, „Principal component analysis: a review and recent developments“, *Philos. Transact. A Math. Phys. Eng. Sci.*, let. 374, št. 2065, apr. 2016.

[4] T. Abdelaal *in sod.*, „A comparison of automatic cell identification methods for single-cell RNA sequencing data“, *Genome Biol.*, let. 20, št. 1, str. 194, sep. 2019.

[5] F. Ma in M. Pellegrini, „ACTINN: automated identification of cell types in single cell RNA sequencing“, *Bioinformatics*, let. 36, št. 2, str. 533–538, jan. 2020.

[6] A. W. Zhang *in sod.*, „Probabilistic cell-type assignment of single-cell RNA-seq for tumor microenvironment profiling“, *Nat. Methods*, let. 16, št. 10, str. 1007–1015, okt. 2019.

[7] A. M. Turing, „Computing Machinery and Intelligence“, *Mind*, let. 59, št. October, str. 433–60, 1950.

[8] B. Darrach, „Meet Shaky, the First Electronic Person: The Fearsome Reality of a Machine with a Mind of Its Own“, *LIFE*, New York, str. 58B, 1970.

[9] C. Cortes in V. Vapnik, „Support-vector networks“, *Mach. Learn.*, let. 20, št. 3, str. 273–297, sep. 1995.

[10] D. E. Rumelhart, G. E. Hinton in R. J. Williams, „Learning representations by back-propagating errors“, *Nature*, let. 323, št. 6088, str. 533–536, okt. 1986.

[11] M. D. Zeiler, „ADADELTA: An Adaptive Learning Rate Method“, *ArXiv12125701 Cs*, dec. 2012.

[12] G. E. Hinton in R. R. Salakhutdinov, „Reducing the Dimensionality of Data with Neural Networks“, *Science*, let. 313, št. 5786, str. 504–507, jul. 2006.

[13] D. Talwar, A. Mongia, D. Sengupta in A. Majumdar, „AutoImpute: Autoencoder based imputation of single-cell RNA-seq data“, *Sci. Rep.*, let. 8, št. 1, str. 1–11, nov. 2018.

[14] D. P. Kingma in M. Welling, „Auto-Encoding Variational Bayes“, *ArXiv13126114 Cs Stat*, maj 2014.

[15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever in R. Salakhutdinov, „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“, *J. Mach. Learn. Res.*, let. 15, str. 1929–1958, 2014.

[16] D. P. Kingma in J. Ba, „Adam: A Method for Stochastic Optimization“, *ArXiv14126980 Cs*, jan. 2017.

[17] F. Chollet *in sod.*, *Keras*. 2015.

[18] Martín Abadi *idr.*, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015.

[19] S. van der Walt, S. C. Colbert in G. Varoquaux, „The NumPy Array: A Structure for Efficient Numerical Computation“, *Comput. Sci. Eng.*, let. 13, št. 2, str. 22–30, mar. 2011.

[20] W. McKinney, „Pandas: a foundational Python library for data analysis and statistics“, *Python High Perform. Sci. Comput.*, let. 14, 2011.

[21] P. Virtanen *in sod.*, „SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python“, *Nat. Methods*, 2020.

[22] F. A. Wolf, P. Angerer in F. J. Theis, „SCANPY: large-scale single-cell gene expression data analysis“, *Genome Biol.*, let. 19, št. 1, str. 15, feb. 2018.

[23] J. D. Hunter, „Matplotlib: A 2D graphics environment“, *Comput. Sci. Eng.*, let. 9, št. 3, str. 90–95, 2007.

[24] O. Franzén, L.-M. Gan in J. L. M. Björkegren, „PanglaoDB: a web server for exploration of mouse and human single-cell RNA sequencing data“, *Database*, let. 2019, jan. 2019.

[25] G. X. Y. Zheng *in sod.*, „Massively parallel digital transcriptional profiling of single cells“, *Nat. Commun.*, let. 8, str. 14049, 16 2017.

[26] S. Hrvatin *in sod.*, „Single-cell analysis of experience-dependent transcriptomic states in the mouse visual cortex“, *Nat. Neurosci.*, let. 21, št. 1, str. 120–129, 2018.

[27] The Tabula Muris Consortium, „Single-cell transcriptomics of 20 mouse organs creates a Tabula Muris“, *Nature*, let. 562, št. 7727, str. 367–372, okt. 2018.

[28] R. Chen, X. Wu, L. Jiang in Y. Zhang, „Single-Cell RNA-Seq Reveals Hypothalamic Cell Diversity“, *Cell Rep.*, let. 18, št. 13, str. 3227–3241, 28 2017.

[29] N. D. Köhler, M. Büttner in F. J. Theis, „Deep learning does not outperform classical machine learning for cell-type annotation“, *bioRxiv*, str. 653907, maj 2019.

[30] R. R. Ricardo-Gonzalez *in sod.*, „Tissue signals imprint ILC2 identity with anticipatory function“, *Nat. Immunol.*, let. 19, št. 10, str. 1093–1099, okt. 2018.

[31] F. Bre, J. M. Gimenez in V. D. Fachinotti, „Prediction of wind pressure coefficients on building surfaces using artificial neural networks“, *Energy Build.*, let. 158, str. 1429–1441, jan. 2018.

7 LITERATURA

I. I. Goodfellow, Y. Bengio, in A. Courville, Deep learning. Cambridge, Massachusetts: The MIT Press, 2016.

II. F. Chollet, Deep learning with Python. Shelter Island, New York: Manning Publications Co, 2018.

III. M. Nielsen, Neural networks and deep learning. Determination Press, 2015

PRILOGA A

deep/analysis.py

```
import matplotlib.pyplot as plt
import numpy as np
from seaborn import distplot
from sklearn import metrics
from scipy.special import softmax
from pandas import Series
import math

from .. import backend


def plot_marker_genes(markers, partially_dense=False):
    """
    Draws a bar chart of marker genes per cell type.
    """
    by_type = backend.sort_markers_by_type(markers)

    hist_data = []
    for i, cell_type in enumerate(by_type):
        gene_n = len(by_type[cell_type])
        if partially_dense:
            gene_n = round(np.log2(gene_n))
        hist_data.append(gene_n)

    hist_mean = np.mean(hist_data)
    hist_data = sorted(hist_data, reverse=True)
    ticks = list(range(len(by_type)))

    plt.bar(ticks, hist_data, width=1.0)
    plt.axhline(hist_mean, c='r', label='Mean ({0})'.format(int(np.round(hist_mean))))
    plt.legend()

    title = 'Number of marker genes per cell type'
    if partially_dense:
        title = 'Number of nodes in the partially dense layer per cell type'
        title += '\nSize of partially-dense layer: {0}'.format(int(sum(hist_data)))
    plt.title(title)


def get_predictions(cell_activations, markers):
    """
    Returns the predicted classes.
    """
    cell_types = backend.get_cell_types(markers)

    top_activations = backend.get_top_activated_indices(1, cell_activations)
    predictions = backend.index_to_cell_type(top_activations, cell_types)

    return [prediction[0] for prediction in predictions]
```

```python
def get_results(labels: list, cell_activations: list, markers: list, aliases: dict) -> float:
    """
    Compares the real labels to the cell type activations.
    :param labels: Data labels
    :param cell_activations: Activations of the Marker Layer nodes
    :param markers: list of used markers
    :param aliases: dict of `label : name_in_marker_db` aliases
    :return
    """

    predictions = get_predictions(cell_activations, markers)
    by_type = backend.sort_markers_by_type(markers)

    correct_types = {}
    correct = 0
    n = len(cell_activations)
    for i, prediction in enumerate(predictions):
        label = labels[i]

        if label not in correct_types.keys():
            correct_types[label] = 0

        if prediction == label:
            correct_types[label] += 1
            correct += 1

        elif label in aliases.keys():
            if aliases[label] == prediction:
                correct_types[label] += 1
                correct += 1

    print(f"Correct predictions: {correct} out of {n} ({round(100 * (correct / n), 2)}%)")

    label_counts = labels.value_counts()
    for c_type in correct_types:
        if c_type in aliases.keys():
            n_markers = len(by_type[aliases[c_type]])
        else:
            n_markers = len(by_type[c_type])

        c = round(100 * (correct_types[c_type] / label_counts[c_type]), 2)
        print("\t{}: {}% ({}/{}) | Markers: {}".format(
            c_type, c, correct_types[c_type], label_counts[c_type], n_markers))

    return correct / n


def plot_model_history(history, baseline_val_acc=None, supervised=False, labelled_training=False):
    """
    Draws a model's training history.
    :param baseline_val_acc: accuracy of the baseline model on the validation set
    :param history: a Keras history object
    :param supervised: was the model supervised
```

```
    """
    fig, (ax1, ax2) = plt.subplots(2, 1, sharex='all', figsize=(7, 7), dpi=80)
    output_loss = history.history['output_loss']

    accuracy = history.history['cell_activations_marker_prediction_metric']
    accuracy = [i * 100 for i in accuracy]

    epochs = range(1, len(output_loss) + 1)

    if supervised:
        output_loss = [i * 100 for i in output_loss]
        marker_loss = history.history['cell_activations_loss']
        ax1.plot(epochs, marker_loss, 'b-,', label='Cell type prediction loss')

    ax1.plot(epochs, output_loss, 'b--', label='Reconstruction loss')

    if labelled_training:
        ax2.plot(epochs, accuracy, 'g--', label='Training accuracy')

    if 'val_loss' in history.history.keys():
        val_output_loss = history.history['val_output_loss']

        val_accuracy = history.history['val_cell_activations_marker_prediction_metric']
        val_accuracy = [i * 100 for i in val_accuracy]

        if supervised:
            val_output_loss = [i * 100 for i in val_output_loss]
            val_marker_loss = history.history['val_cell_activations_loss']
            ax1.plot(epochs, val_marker_loss, 'r-', label='Validation cell type prediction loss')

        ax1.plot(epochs, val_output_loss, 'r--', label='Validation reconstruction loss')
        ax1.set_title('Training and validation loss')

        ax2.plot(epochs, val_accuracy, 'g-', label='Validation accuracy')

        if labelled_training:
            ax2.set_title('Training, validation and baseline accuracy')
        else:
            ax2.set_title('Validation and baseline accuracy')

        if baseline_val_acc is not None:
            baseline_val_acc = baseline_val_acc * 100 if baseline_val_acc <= 1. else
baseline_val_acc
            ax2.axhline(baseline_val_acc, c='r', label=f'Baseline accuracy
({round(baseline_val_acc, 2)}%)')
    else:
        ax1.set_title('Training loss per epoch')
        ax2.set_title('Training accuracy per epoch')

    fig.text(0.04, 0.5, 'Accuracy (%)                                                    Loss',
va='center',
             rotation='vertical')
    plt.xlabel('Epochs')
    ax1.legend()
    ax2.legend()
```

```
    plt.show()


def plot_activation_distribution(cell_activations, markers, title=''):
    """
    Plots the average cell type activation.
    :param cell_activations: Marker layer node activations
    :param markers: list of used markers
    :param title: graph title
    """
    n_cell_types = len(backend.get_cell_types(markers))
    sorted_indices = backend.get_top_activated_indices(n_cell_types, cell_activations)

    sorted_activations = []
    for i, cell in enumerate(cell_activations):
        sorted_activations.append([cell[j] for j in sorted_indices[i]])

    average = [np.mean(column) for column in np.transpose(sorted_activations)]

    distplot(average).set_title(title)


def get_label_colours(labels):
    """
    Generates a dictionary of colours for each label
    :param labels: list of labels
    :return: dictionary of label : colour
    """
    colours = {}
    for label in labels.unique():
        colour = (backend.get_random_colour())
        colours.update({label: colour})

    return colours


def plot_label_colours(colours):
    """
    Plots a list of colours
    :param colours:
    """
    if type(colours) == dict:
        colours = colours.values()

    fig = plt.figure()
    ax = fig.add_subplot(111)

    x = list(range(0, len(colours) * 2, 2))
    y = [0] * len(x)
    c = list(colours)

    ax.scatter(x, y, s=400, c=c)
    ax.set_title('Cell type colours')
    ax.set_yticklabels([])
```

```
    ax.set_xticklabels([])


def get_top_activations(n, cell_activations):
    """
    Returns cell activations with only the top n activations
    :param n: top number of cell types to return
    :param cell_activations: list of cell activations
    """
    top_indices = backend.get_top_activated_indices(n, cell_activations)
    top_activations = [np.zeros(len(x)) for x in cell_activations]
    for i, cell_indices in enumerate(top_indices):
        for type_index in cell_indices:
            top_activations[i][type_index] = cell_activations[i][type_index]

    return top_activations


def draw_embedding(x, y, model, colours=None, alpha=1.0, graph_title=''):
    """
    Draws a scatter plot of the provided embedding model
    :param x: Data x
    :param y: Data labels
    :param model: embedder e.g. tSNE or PCA
    :param colours: label colours
    :param alpha: Node alpha
    :param graph_title: Graph title
    """
    if colours is None:
        colours = get_label_colours(y)

    model_out = model.fit_transform(x)

    plt.figure(figsize=(8, 8), dpi=80)
    for i, point in enumerate(model_out):
        plt.scatter(point[0], point[1], color=colours[y.iloc[i]], label=y.iloc[i],
                    alpha=alpha, edgecolors='black', linewidths=1.0)

    # Create a legend
    plt_handles, plt_labels = plt.gca().get_legend_handles_labels()
    handles, labels = backend.get_graph_labels(plt_handles, plt_labels)

    for handle in handles:
        handle.set_alpha(1.0)

    plt.legend(handles, labels, bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)

    plt.title(graph_title)
    plt.axis('off')
    plt.show()


def compare_embeddings(data_1, data_2, model, colours=None, alpha=1.0, graph_titles=None):
    """
    Draws a scatter plot of the provided embedding model
```

```
    :param data_1: First data set -- list of the data's x and y
    :param data_2: Second data set -- list of data's x and y
    :param model: embedder e.g. tSNE or PCA
    :param colours: label colours
    :param alpha: Node alpha
    :param graph_titles: list of graph titles
    """
    x1, y1 = data_1[0], data_1[1]
    x2, y2 = data_2[0], data_2[1]

    if colours is None:
        colours = get_label_colours(y1.append(y2))

    model_out_1 = model.fit_transform(x1)
    model_out_2 = model.fit_transform(x2)

    fig, (ax1, ax2) = plt.subplots(ncols=2, sharex=True, sharey=True, figsize=(16, 16), dpi=120)
    ax1.set(aspect='equal')
    ax2.set(aspect='equal')

    for i, point in enumerate(model_out_1):
        ax1.scatter(point[0], point[1], color=colours[y1.iloc[i]], label=y1.iloc[i],
                    alpha=alpha, edgecolors='black', linewidths=1.0)
    for i, point in enumerate(model_out_2):
        ax2.scatter(point[0], point[1], color=colours[y2.iloc[i]], label=y2.iloc[i],
                    alpha=alpha, edgecolors='black', linewidths=1.0)

    # Create a legend
    plt_handles, plt_labels = plt.gca().get_legend_handles_labels()
    handles, labels = backend.get_graph_labels(plt_handles, plt_labels)

    for handle in handles:
        handle.set_alpha(1.0)

    fig.legend(handles, labels, ncol=len(labels), loc=8, bbox_to_anchor=(0.425, 0.1))

    ax1.set_title(graph_titles[0])
    ax1.set_axis_off()

    ax2.set_title(graph_titles[1])
    ax2.set_axis_off()

    plt.subplots_adjust(bottom=0., hspace=0., wspace=0.25)
    plt.show()


def draw_comparison(old, new, model, colours=None, graph_title=''):
    """
    Draws a combined graph using both data sets
    :param old: a list of old data (data, labels)
    :param new: a list of new data (data, labels)
    :param model: embedder e.g. tSNE or PCA
    :param colours: label colours
    :param graph_title: Graph title
    """
```

```
    if colours is None:
        colours = get_label_colours(old[1])

    x = np.concatenate((old[0], new[0]), axis=0)
    model_out = model.fit_transform(x)

    plt.figure(figsize=(8, 8), dpi=80)
    for i, point in enumerate(model_out):
        if i < len(old[0]):
            plt.scatter(point[0], point[1], color=colours[old[1].iloc[i]], label=old[1].iloc[i],
alpha=0.1)
        else:
            index = i - len(old[0])
            plt.scatter(point[0], point[1], color=colours[new[1].iloc[index]],
label=new[1].iloc[index],
                        alpha=1.0, edgecolors='black', linewidths=1)

    # Create a legend
    plt_handles, plt_labels = plt.gca().get_legend_handles_labels()
    handles, labels = backend.get_graph_labels(plt_handles, plt_labels)

    for handle in handles:
        handle.set_edgecolors('black')
        handle.set_alpha(1.0)

    plt.legend(handles, labels, bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)

    plt.title(graph_title)
    plt.axis('off')
    plt.show()


def draw_confusion_matrix(labels, cell_activations, markers, aliases, title=None,
cmap=plt.cm.Greens):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if not title:
        title = 'Confusion matrix'

    cell_types = backend.get_cell_types(markers)
    top_activations = backend.get_top_activated_indices(1, cell_activations)

    predictions = backend.index_to_cell_type(top_activations, cell_types)
    predictions = Series([p[0] for p in predictions])

    y_true = labels
    y_true = y_true.apply(lambda x: aliases[x] if x in aliases.keys() else x)
    unique_classes = sorted(y_true.append(predictions).unique())

    cm = metrics.confusion_matrix(y_true, predictions)

    fig, ax = plt.subplots(figsize=(10, 10), dpi=80)
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
```

```
    ax.figure.colorbar(im, ax=ax)

    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           xticklabels=unique_classes, yticklabels=unique_classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], 'd'),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")


def draw_roc(labels, cell_activations, markers, aliases):
    """
    Returns the average ROC AUC score and, if defined, draws an ROC graph for each class.
    """
    probs = softmax(cell_activations, axis=1)
    evaluation = backend.get_class_evaluation(labels, cell_activations, markers, aliases)

    by_type = backend.sort_markers_by_type(markers)
    types = list(by_type.keys())
    y_true = backend.one_hot_encode(labels, markers, aliases)

    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    n_classes = len(types)

    for i in range(n_classes):
        fpr[i], tpr[i], _ = metrics.roc_curve(y_true[:, i], probs[:, i])
        roc_auc[i] = metrics.auc(fpr[i], tpr[i])

    n = 0
    used = []
    to_draw = []

    for i, c_type in enumerate(types):
        score = roc_auc[i]
        if not np.isnan(score):
            n += 1
            used.append(score)
            to_draw.append([fpr[i], tpr[i], score, c_type])

    # Drawing the plots
    n_columns = 3
    n_rows = int(math.ceil(len(to_draw) / n_columns))
```

```
    fig, ax = plt.subplots(nrows=n_rows, ncols=n_columns)
    fig.set_figheight(15)
    fig.set_figwidth(15)
    plt.subplots_adjust(top=0.6, bottom=0.01, hspace=0.2, wspace=0.2)

    k = 0
    for i, col in enumerate(ax):
        for j, row in enumerate(col):
            if k > len(to_draw):
                row.remove()
                continue

            plot = to_draw[k]
            row.plot(plot[0], plot[1], label='ROC curve (area = %0.2f)' % plot[2])
            row.plot([0, 1], [0, 1], 'k--')

            row.set_aspect(1.)
            row.set_xlim([0.0, 1.0])
            row.set_ylim([0.0, 1.05])

            f1 = evaluation[plot[3]]['f1']
            row.plot([], [], ' ', label=f'F1 score: {round(f1, 3)}')

            row.set_xlabel('False Positive Rate')
            row.set_ylabel('True Positive Rate')
            row.title.set_text(f'ROC curve of {plot[3]}')
            row.legend(loc="lower right")

            k += 1

    plt.show()
```

deep/data.py

```
from .. import backend
import anndata as ann
import numpy as np
import pandas as pd


def load_h5ad(path: str):
    """
    Loads a data set from the h5ad format as an AnnData object
    """
    data_set = ann.read_h5ad(path)
    return data_set


def load_markers(marker_path: str, species: str) -> pd.DataFrame:
    """
    Loads marker genes of the specified species.
    :param marker_path: The path to the tab separated file
    :param species: "Mouse", "Human" etc.
    :return: List of markers
    """

    markers_db = pd.read_csv(marker_path, delimiter='\t')
    markers = markers_db[markers_db['Organism'] == species]

    return markers


def get_mutual_markers(data_sets: list, markers_db: list) -> list:
    """
    :param data_sets: list of data sets
    :param markers_db: a list of markers
    :return: an intersection of markers used in the provided data set(s).
    """
    gene_sets = [data_set.var_names for data_set in data_sets]

    mutual = set(backend.get_used_markers(genes=gene_sets[0], markers=markers_db))
    for genes in gene_sets[1:]:
        mutual.intersection_update(backend.get_used_markers(genes=genes, markers=markers_db))

    return list(mutual)


def drop_cell_types(n, markers):
    """
    Removes every cell type with less <= n used marker genes
    """
    by_type = backend.sort_markers_by_type(markers)
    types_to_drop = []
    for i, cell_type in enumerate(by_type):
        gene_n = len(by_type[cell_type])
        if gene_n <= n:
```

```
            types_to_drop.append(cell_type)

    to_drop = []
    for marker in markers:
        c_type = marker[2]
        if c_type in types_to_drop:
            to_drop.append(marker)

    return [marker for marker in markers if marker not in to_drop]


def drop_rows(data, row_names: list):
    """
    Drops rows by label.
    """
    to_keep = backend.get_rows_to_keep(row_names, data.obs['labels'])

    n_dropped = data.shape[0] - len(to_keep)
    data = data[to_keep, :]

    print("Dropped {0} cell(s).".format(n_dropped), "New shape:", data.shape)

    return data


def drop_unused_genes(data, markers, sort_columns=True):
    """
    Drops every column (gene) not found in the marker data set and sorts the columns by name.
    """
    sorted_by_type = backend.sort_markers_by_type(markers)
    used_genes = backend.get_used_genes(sorted_by_type)

    n_dropped = data.shape[1] - len(used_genes)
    data = data[:, used_genes]

    if sort_columns:
        data = data[:, np.argsort(data.var_names)]

    print("Dropped {0} gene(s).".format(n_dropped), "New shape:", data.shape)
    return data


def check_labels(data_sets: list, markers: list, aliases: dict, throw_exception=True) -> None:
    """
    Checks for any cell types (labels) not found in the marker data set.
    """
    unknown = []
    used_types = backend.get_cell_types(markers)

    for data_set in data_sets:
        labels = data_set.obs['labels']
        for unknown_label in backend.check_for_unknown(labels.unique(), used_types, aliases):
            if unknown_label not in unknown:
                unknown.append(unknown_label)
```

```
    if len(unknown) != 0:
        if throw_exception:
            raise NameError("Unknown cell type(s)!", unknown)
        else:
            print("====UNKNOWN CELL TYPES====")
            print(unknown)
            print("=========================")


def check_shape(data_sets: list, throw_exception=True) -> None:
    """
    Checks if the there are the same number of genes
    :param data_sets: list of DataFrames to check
    :param throw_exception: throw exception if shapes do not match
    """
    num_genes = data_sets[0].shape[1]
    for data_set in data_sets[1:]:
        if num_genes != data_set.shape[1]:
            if throw_exception:
                raise ValueError("Column lengths do not match!", num_genes, len(data_set.columns))
            else:
                print("COLUMN LENGTHS DO NOT MATCH!", num_genes, len(data_set.columns))

    genes = data_sets[0].var_names.values
    for data_set in data_sets[1:]:
        if np.sum(genes == data_set.var_names.values) - len(genes) != 0:
            if throw_exception:
                raise ValueError("Columns are not in the same order!")
            else:
                print("Columns are not in the same order!")
```

deep/model.py

```
from .. import backend
from ..backend import Partial
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras import initializers as inits

import pandas as pd
import numpy as np
from scipy.sparse import lil_matrix
from anndata import AnnData

from .data import drop_unused_genes


def build_model(data, markers, bottleneck_dim=25, intermediate_dim=100, dropout_n=0.1,
                activation='relu', loss='mse', optimizer='adam', supervised=False):
    """
    Builds the AutoEncoder model.
    :param data: DataFrame to train/test on
    :param markers: list of used markers
    :param bottleneck_dim: number of bottleneck nodes
    :param intermediate_dim: number of dense layer nodes
    :param dropout_n: dropout rate (from 0 to 1.0)
    :param activation: activation function
    :param loss: loss function
    :param optimizer: optimizer function
    :param supervised: should the model use a categorical loss on the marker layer?
    :return: AutoEncoder, marker and encoder models
    """
    by_type = backend.sort_markers_by_type(markers)
    input_dim = data.shape[1]
    marker_dim = len(by_type)
    partial_dim = backend.get_partially_dense_size(by_type)

    partially_dense_mask = backend.get_partially_dense_mask(by_cell_type=by_type,
genes=data.var_names)
    weight_mask = backend.get_marker_mask(by_cell_type=by_type)

    # -- Model --
    input_layer = Input(shape=(input_dim,))

    partial_input = Partial(partial_dim, weight_mask=partially_dense_mask, use_bias=True,
                            # kernel_initializer=inits.RandomNormal(mean=0.5, stddev=0.5),
                            # kernel_initializer=inits.ones(),
                            activation=activation)(input_layer)

    marker_layer = Partial(marker_dim, weight_mask=weight_mask, use_bias=True,
                           # kernel_initializer=inits.RandomNormal(mean=0.5, stddev=0.5),
                           # kernel_initializer=inits.ones(),
                           activation=activation, name='cell_activations')(partial_input)

    dense_in_1 = Dense(intermediate_dim, activation=activation)(marker_layer)
```

```
    bottleneck_layer = Dense(bottleneck_dim, activation=activation, name='Bottleneck')(dense_in_1)
    dense_out_1 = Dense(intermediate_dim, activation=activation)(bottleneck_layer)

    dropout = Dropout(rate=dropout_n)(dense_out_1)
    output_layer = Dense(input_dim, activation=activation, name='output')(dropout)
    # --------
    autoencoder_model = Model(input_layer, [marker_layer, output_layer])
    marker_model = Model(input_layer, marker_layer)
    encoder_model = Model(input_layer, bottleneck_layer)

    if supervised:
        autoencoder_model.compile(loss={'cell_activations': backend.marker_loss, 'output': loss},
                                  loss_weights={'cell_activations': 1., 'output': 1.},
                                  metrics={'cell_activations': backend.celltype_accuracy},
                                  optimizer=optimizer)
    else:
        autoencoder_model.compile(loss={'cell_activations': backend.null_loss, 'output': loss},
                                  loss_weights={'cell_activations': 1., 'output': 1000.},
                                  metrics={'cell_activations': backend.celltype_accuracy},
                                  optimizer=optimizer)
    marker_model.compile(loss=loss, optimizer=optimizer)
    encoder_model.compile(loss=loss, optimizer=optimizer)

    return autoencoder_model, marker_model, encoder_model


def train_model(model, data, markers, marker_aliases, epochs,
                validation_data=None, batch_size=256, verbose=1, callbacks=None):
    """
    Trains the Keras model.
    :return: a Keras train history object
    """
    if validation_data is not None:
        validation_labels = validation_data.obs['labels']
        val_labels_one_hot = backend.one_hot_encode(validation_labels, markers, marker_aliases)

        dense_val = validation_data.X.todense()
        validation_data = (dense_val, {'cell_activations': val_labels_one_hot, 'output':
dense_val})

    labels_one_hot = backend.one_hot_encode(data.obs['labels'], markers, marker_aliases)
    dense_data = data.X.todense()
    history = model.fit(dense_data, {'cell_activations': labels_one_hot, 'output': dense_data},
                        epochs=epochs, batch_size=batch_size,
                        validation_data=validation_data,
                        callbacks=callbacks,
                        verbose=verbose)

    return history


def test_model(model, data, markers, aliases, verbose=0):
    """
    Evaluates the model on the given data
    :return: the data's loss score
```

```
    """
    labels_one_hot = backend.one_hot_encode(data.obs['labels'], markers, aliases)
    results = model.evaluate(data.X, {'cell_activations': labels_one_hot, 'output': data.X},
verbose=verbose)

    print("Test reconstruction loss:", round(results[2], 8))
    print("Test prediction accuracy:", round(results[3] * 100, 3), "%")

    return results


def get_baseline(data, markers):
    by_type = backend.sort_markers_by_type(markers)
    matrix = lil_matrix((data.shape[0], len(by_type)))

    for i, c_type in enumerate(by_type):
        column_matrix = len(data[:, by_type[c_type]].X.shape) > 1

        if column_matrix:
            result = data[:, by_type[c_type]].X.sum(axis=1)
            result = result / len(by_type[c_type])
            matrix[:, i] = np.reshape(result, matrix[:, i].shape)
        else:
            matrix[:, i] = np.zeros(matrix[:, i].shape)

    var = pd.DataFrame(index=[c_type for c_type in by_type])
    return AnnData(X=matrix.tocsc(), obs=data.obs, var=var)
```

deep/preprocessing.py

```
from .. import backend
import pandas as pd
import numpy as np


def preprocess_data(data, train=0.7, validation=0.15, test=0.15, splits=None, normalize=True):
    """
    Shuffles, log10 transforms and splits the data into train, validation and test sets.
    """
    if splits is None:
        splits = 1

    train_index, validation_index, test_index = backend.get_data_splits(data.shape[0], train,
validation, test)
    data = backend.shuffle_data(data, axis=0)

    split_size = data.shape[0] // splits
    if normalize:
        data = backend.normalize_data(data.copy(), split_size)

    train = data[:train_index]
    validation = data[validation_index:test_index]
    test = data[train_index:]

    return train, validation, test


def mark_as_unlabelled(data):

    if 'labels' in data.obs_keys():
        data.obs['labels'] = pd.Series(np.repeat(-1, data.shape[0]),
                                       index=data.obs['labels'].index)
    else:
        data.obs.insert(0, 'labels', np.repeat(-1, data.shape[0]), True)
    return data
```

backend/analysis.py

```
import pickle
from random import uniform, seed
from time import time
from collections import OrderedDict
from saly import backend
import pandas as pd
from sklearn import metrics
import numpy as np


def get_top_activated_indices(n: int, cell_type_activations: list) -> list:
    """
    Returns a list of indices of top n activations
    :param n: number of top activations to return
    :param cell_type_activations: list of Marker Layer node activations
    :return: list of n indices
    """
    tops = []
    for cell in cell_type_activations:
        top = cell.argsort()[-n:][::-1]
        tops.append(top)

    return tops


def index_to_cell_type(indices, cell_types):
    """
    Converts Marker Layer activation indices to cell types
    :param indices: list of Marker Layer activation indices
    :param cell_types: list of cell types in the marker layer
    :return: list of activated cell types
    """
    activated_types = []

    for cell in indices:
        activated_types.append([cell_types[i] for i in cell])

    return activated_types


def get_random_colour():
    """
    Generates random RGB values.
    """
    seed(time())
    pastel_factor = uniform(0.0, 0.8)
    return [(x + pastel_factor) / (1.0 + pastel_factor) for x in [uniform(0, 1.0) for _ in [1, 2,
3]]]


def save_label_colours(colours: dict, path: str) -> str:
    """
```

```
    Saves a dictionary of label : colour into a pickle file
    :param colours: a dictionary of label : colour pairs
    :param path: path to save folder
    :return: file location
    """
    name = path + 'label_colours.pickle'

    pickle_out = open(name, "wb")
    pickle.dump(colours, pickle_out)
    pickle_out.close()

    return name


def load_label_colours(path: str) -> dict:
    """
    Loads a dictionary from a pickle file
    :param path: path to file
    :return: loaded dictionary
    """
    pickle_in = open(path, "rb")
    colours = pickle.load(pickle_in)

    return colours


def get_graph_labels(handles, plt_labels):
    by_label = OrderedDict(zip(plt_labels, handles))
    by_label = OrderedDict(sorted(by_label.items()))

    return by_label.values(), by_label.keys()


def get_class_evaluation(labels, cell_activations, markers, aliases):
    cell_types = backend.get_cell_types(markers)
    top_activations = backend.get_top_activated_indices(1, cell_activations)

    predictions = backend.index_to_cell_type(top_activations, cell_types)
    predictions = pd.Series([p[0] for p in predictions])

    y_true = labels
    y_true = y_true.apply(lambda x: aliases[x] if x in aliases.keys() else x)
    unique_classes = sorted(y_true.append(predictions).unique())
    true_classes = sorted(y_true.unique())

    cm = metrics.confusion_matrix(y_true, predictions)

    fp = cm.sum(axis=0) - np.diag(cm)
    fn = cm.sum(axis=1) - np.diag(cm)
    tp = np.diag(cm)
    tn = cm.sum() - (fp + fn + tp)

    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    f1_score = 2 * (recall * precision) / (recall + precision)
```

```
    results = {}
    for i, cls in enumerate(unique_classes):
        if cls in true_classes:
            result = {'tp': tp[i], 'tn': tn[i], 'fp': fp[i], 'fn': fn[i],
                      'precision': precision[i], 'recall': recall[i], 'f1': f1_score[i]}
            results.update({cls: result})
    return results
```

backend/data.py

```
from collections import OrderedDict


def get_used_markers(genes: list, markers) -> list:
    """
    Returns the marker genes found in the given data set.
    """
    used_markers = []
    for i, gene_name in enumerate(genes):
        used = markers.loc[markers['Name'] == gene_name]
        if not used.empty:
            [used_markers.append(tuple(marker)) for marker in used.values.tolist()]
    return used_markers


def sort_markers_by_type(markers: list) -> OrderedDict:
    """
    Returns an OrderedDictionary of every cell type and its
    corresponding genes found in the given array of markers.
    :param markers: list of used markers
    """
    by_type = OrderedDict()
    for marker in markers:
        gene_name = marker[1]
        c_type = marker[2]

        # check if the key already exists
        if c_type in by_type:
            # check if the gene is already in the array
            if gene_name not in by_type[c_type]:
                by_type[c_type].append(gene_name)

        else:  # if not, create a new entry
            by_type[c_type] = [gene_name]
    return by_type


def get_used_genes(markers_by_type):
    """
    Returns a (unique) array of genes used in the specified array.
    :param markers_by_type
    :return
    """
    used_genes = []
    for c_type in markers_by_type:
        genes = markers_by_type[c_type]
        for gene in genes:
            if gene not in used_genes:
                used_genes.append(gene)
    return used_genes
```

```python
def get_cell_types(markers: list) -> list:
    """
    Returns an array of cell types given an array of used markers.
    :param markers: used markers
    :return: 2D array of cell_type=>[genes]
    """
    used_types = []
    for marker in markers:
        c_type = marker[2]
        if c_type not in used_types:
            used_types.append(c_type)
    return used_types


def get_rows_to_drop(labels_to_drop: list, labels: list) -> list:
    """
    Returns a list of row indices to drop from a list of row labels.
    :return: list of indices
    """
    return [i for i, c_type in enumerate(labels) if c_type in labels_to_drop]


def get_rows_to_keep(labels_to_drop: list, labels: list) -> list:
    """
    Returns a list of row indices to keep from a list of row labels.
    :return: list of indices
    """
    return [i for i, c_type in enumerate(labels) if c_type not in labels_to_drop]


def check_for_unknown(labels: list, cell_types: list, aliases: dict) -> list:
    """
    Returns cell types not found in either of the provided lists.
    :param labels: list of labels
    :param cell_types: list of cell types
    :param aliases: dictionary of aliases
    :return: unknown labels
    """
    unknown = []
    known_aliases = list(aliases.keys())
    for c_type in labels:
        if c_type not in cell_types and c_type not in known_aliases:
            if c_type not in unknown:
                unknown.append(c_type)

    return unknown
```

backend/model.py

```
from .data import sort_markers_by_type

import tensorflow as tf
import numpy as np
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Layer
from tensorflow.keras import activations, initializers, regularizers, constraints
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.metrics import categorical_accuracy
from tensorflow.keras.layers import InputSpec


def get_partially_dense_size(by_type):
    """
    Returns the number of nodes in the partially dense layer.
    """
    nodes = []
    for i, cell_type in enumerate(by_type):
        gene_n = len(by_type[cell_type])
        gene_n = round(np.log2(gene_n))
        nodes.append(gene_n)

    return int(np.sum(nodes))


def get_partially_dense_mask(by_cell_type, genes):
    """
    Creates a binary mask for the partially dense layer.
    """
    node_dim = get_partially_dense_size(by_cell_type)
    mask = np.zeros(shape=(len(genes), node_dim))

    i = 0
    for cell_type in by_cell_type:
        marker_genes = by_cell_type[cell_type]
        N = len(marker_genes)
        n = int(round(np.log2(N)))

        for node in range(n):
            for gene in by_cell_type[cell_type]:
                gene_index = genes.get_loc(gene)
                mask[gene_index][i] = 1.0
            i += 1

    return mask


def get_marker_mask(by_cell_type):
    """
    Creates a binary mask for the marker layer.
    :param by_cell_type: Markers sorted by cell type
    """
```

```
    node_dim = get_partially_dense_size(by_cell_type)
    mask = np.zeros(shape=(node_dim, len(by_cell_type)))

    i = 0
    for c, cell_type in enumerate(by_cell_type):
        marker_genes = by_cell_type[cell_type]
        N = len(marker_genes)
        n = int(round(np.log2(N)))

        for node in range(n):
            mask[i][c] = 1.0
            i += 1
    return mask


def get_weight_mask(shape, by_cell_type, genes):
    """
    Deprecated.
    Creates a binary mask for the marker layer
    :param shape: shape of the matrix (n of cell types, n of genes)
    :param by_cell_type: Markers sorted by cell type
    :param genes: list of used genes (in the same order as in the data)
    """
    mask = np.zeros(shape=shape)
    for i, cell_type in enumerate(by_cell_type):
        for gene in by_cell_type[cell_type]:
            gene_index = genes.get_loc(gene)
            mask[i][gene_index] = 1.0
    return mask


def one_hot_encode(labels, markers, aliases):
    """
    One hot encodes a list of marker cell types
    :param labels: Used labels
    :param markers: Used markers
    :param aliases: Cell type aliases (saly.check_labels)
    """
    by_type = sort_markers_by_type(markers)
    types = list(by_type.keys())

    one_hot = np.zeros(shape=(len(labels), len(by_type)))
    for i, label in enumerate(labels):
        if label == -1:
            one_hot[i] = np.repeat(-1, len(by_type))
        else:
            if label in types:
                label_index = types.index(label)
            elif label in aliases.keys():
                label_index = types.index(aliases[label])
            else:
                raise NameError("Unknown cell type!", label)

            one_hot[i][label_index] = 1.0
```

```
    return one_hot


def marker_loss(y_true, y_pred):
    """
    Get the marker cell type activations classification loss
    """
    labelled_mask = tf.where(tf.math.equal(y_true, -1), tf.zeros_like(y_true, bool),
tf.ones_like(y_true, bool))

    labelled_true = tf.boolean_mask(y_true, labelled_mask)
    labelled_pred = tf.boolean_mask(y_pred, labelled_mask)

    # return categorical_crossentropy(y_true, y_pred, from_logits=True)

    labelled_true = tf.reshape(labelled_true, tf.shape(y_true))
    tf.print(y_true.shape)
    tf.print(labelled_true.shape)
    return categorical_crossentropy(labelled_true, labelled_pred, from_logits=True)


def null_loss(y_true, y_pred):
    """
    An empty loss function.
    """
    return 0 * y_true


def celltype_accuracy(y_true, y_pred):
    labelled_mask = tf.where(tf.math.equal(y_true, -1), tf.zeros_like(y_true, bool),
tf.ones_like(y_true, bool))

    labelled_true = tf.boolean_mask(y_true, labelled_mask)
    labelled_pred = tf.boolean_mask(y_pred, labelled_mask)

    return categorical_accuracy(labelled_true, labelled_pred)


class Partial(Layer):

    def __init__(self, units, weight_mask,
                 activation=None,
                 use_bias=True,
                 kernel_initializer='glorot_uniform',
                 bias_initializer='zeros',
                 kernel_regularizer=None,
                 bias_regularizer=None,
                 activity_regularizer=None,
                 kernel_constraint=None,
                 bias_constraint=None,
                 **kwargs):
        if 'input_shape' not in kwargs and 'input_dim' in kwargs:
            kwargs['input_shape'] = (kwargs.pop('input_dim'),)
        super(Partial, self).__init__(**kwargs)
        self.units = units
```

```
        self.activation = activations.get(activation)
        self.use_bias = use_bias
        self.kernel_initializer = initializers.get(kernel_initializer)
        self.bias_initializer = initializers.get(bias_initializer)
        self.kernel_regularizer = regularizers.get(kernel_regularizer)
        self.bias_regularizer = regularizers.get(bias_regularizer)
        self.activity_regularizer = regularizers.get(activity_regularizer)
        self.kernel_constraint = constraints.get(kernel_constraint)
        self.bias_constraint = constraints.get(bias_constraint)
        self.input_spec = InputSpec(min_ndim=2)
        self.supports_masking = True

        weight_mask = tf.convert_to_tensor(weight_mask, dtype=tf.float32)
        self.weight_mask = weight_mask

    def build(self, input_shape):
        assert len(input_shape) >= 2
        input_dim = input_shape[-1]

        self.kernel = self.add_weight(shape=(input_dim, self.units),
                                      initializer=self.kernel_initializer,
                                      name='kernel',
                                      regularizer=self.kernel_regularizer,
                                      constraint=self.kernel_constraint)
        if self.use_bias:
            self.bias = self.add_weight(shape=(self.units,),
                                        initializer=self.bias_initializer,
                                        name='bias',
                                        regularizer=self.bias_regularizer,
                                        constraint=self.bias_constraint)
        else:
            self.bias = None
        self.input_spec = InputSpec(min_ndim=2, axes={-1: input_dim})
        self.built = True

    def call(self, inputs):
        conns = tf.multiply(self.kernel, self.weight_mask)
        output = K.dot(inputs, conns)
        if self.use_bias:
            output = K.bias_add(output, self.bias, data_format='channels_last')
        if self.activation is not None:
            output = self.activation(output)
        return output

    def compute_output_shape(self, input_shape):
        assert input_shape and len(input_shape) >= 2
        assert input_shape[-1]
        output_shape = list(input_shape)
        output_shape[-1] = self.units
        return tuple(output_shape)

    def get_config(self):
        config = {
            'units': self.units,
            'activation': activations.serialize(self.activation),
```

```
            'use_bias': self.use_bias,
            'kernel_initializer': initializers.serialize(self.kernel_initializer),
            'bias_initializer': initializers.serialize(self.bias_initializer),
            'kernel_regularizer': regularizers.serialize(self.kernel_regularizer),
            'bias_regularizer': regularizers.serialize(self.bias_regularizer),
            'activity_regularizer':
                regularizers.serialize(self.activity_regularizer),
            'kernel_constraint': constraints.serialize(self.kernel_constraint),
            'bias_constraint': constraints.serialize(self.bias_constraint)
        }
        base_config = super(Partial, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))
```

backend/preprocessing.py

```
import numpy as np
from scipy import sparse
import scanpy as scpy


def get_data_splits(n: int, train: float, validation: float, test: float) -> (int, int, int):
    """
    Splits the data into train, validation and test sets.
    :param n: length of array to split
    :param train: % of train data
    :param validation: % of validation data
    :param test: % of test data
    :return: Indices of train, validation and test data
    """
    if train + validation + test > 1.0:
        raise ValueError("Data splits sum up to more than 1.0!", train, validation, test)

    train_index = int(n * train)
    validation_index = train_index + int(n * validation)
    test_index = validation_index + int(n * test)

    return train_index, validation_index, test_index


def shuffle_data(data, axis=0):
    """
    Shuffles the data on the given axis.
    """
    if axis == 0:
        idx = np.random.permutation(data.obs['labels'].index)
        data = data[idx, :]
    elif axis == 1:
        idx = np.random.permutation(data.var_names)
        data = data[:, idx]
    else:
        raise ValueError("Axis must be 0 or 1; given ", axis)

    return data


def var(x, axis=None, ddof=0):
    """ Equivalent of np.var that supports sparse and dense matrices. """
    if not sparse.issparse(x):
        return np.var(x, axis, ddof=ddof)

    result = x.multiply(x).mean(axis) - np.square(x.mean(axis))
    result = np.squeeze(np.asarray(result))

    # Apply correction for degrees of freedom
    n = np.prod(x.shape) if axis is None else x.shape[axis]
    result *= n / (n - ddof)
```

```
    return result


def std(x, axis=None, ddof=0):
    """ Equivalent of np.std that supports sparse and dense matrices. """
    return np.sqrt(var(x, axis=axis, ddof=ddof))


def standardize_data(data, mean, std):
    sub = np.subtract(data, mean)
    return np.divide(sub, std, out=np.zeros_like(sub), where=std!=0)


def normalize_data(data, chunk_size=None):
    chunked = chunk_size is not None

    scpy.pp.normalize_total(data, target_sum=1e6)
    scpy.pp.log1p(data)

    """
    c_std = std(data.X, axis=0)
    c_mean = data.X.mean(axis=0).A[0]
    dense = data.X.toarray()
    for chunk, start, end in data.chunked_X(chunk_size):
        dense[start:end] = np.apply_along_axis(standardize_data, 1,
                                               dense[start:end], c_mean, c_std)

    data.X = dense
    """
    return data
```