

Prva laboratorijska vježba iz Dubokog učenja

Gradijenti višeslojnih mreža, uvod u Tensorflow, potpuno povezani duboki modeli, studija slučaja: MNIST

Predmet ove vježbe su unaprijedni duboki modeli za klasifikaciju. Pokazat ćemo da se ti modeli mogu promatrati ili kao višeslojne unaprijedne neuronske mreže ili kao produbljeni logistički modeli koje smo upoznali u nultoj vježbi. Oba pogleda vode na istu programsku izvedbu koja se temelji na optimiranju izglednosti parametara modela. Kako bismo olakšali razvoj i ubrzali eksperimentiranje, proučit ćemo mogućnost automatske diferencijacije koju danas nude brojni programski okviri za numeričku optimizaciju. Posebnu pažnju poklonit ćemo Tensorflowu kao trenutno najnaprednjem alatu te kategorije.

Cilj vježbe je razviti sedam modula: `data`, `fcann2`, `tf_linreg`, `tf_logreg`, `tf_deep`, `ksvm_wrap`, i `mnist_shootout`. Modul `data` će biti nadograđena verzija istoimenog modula iz nulte vježbe. Modul `fcann2` će sadržavati implementaciju dvoslojnog potpuno povezanog modela temeljenog na Numpyjevim primitivima. Organizacijski i izvedbeno, taj modul bi trebao biti vrlo sličan modulu `logreg` iz nulte vježbe. Sljedeća tri modula sadržavat će implementacije triju postupaka strojnog učenja rastuće složenosti, temeljene na okviru Tensorflow. Modul `ksvm_wrap` će umatati klasifikator s jezgrenim ugrađivanjem i potpornim vektorima izведен u modulu `sklearn.svm` biblioteke `scikit-learn` te omogućiti usporedbu s klasifikatorima temeljenima na dubokom učenju. Konačno, modul `mnist_shootout` će usporediti performansu do tada razvijenih klasifikatora na skupu podataka MNIST.

0a. Uvodne napomene o dubokim modelima

Duboki modeli strojnog učenja temelje se na apstraktnim reprezentacijama podataka do kojih dolazimo slijedom naučenih nelinearnih transformacija. U ovoj i sljedećoj vježbi razmatramo duboke modele koji su *diskriminativni* i *unaprijedni*. Diskriminativni model za dani podatak \mathbf{x} na izlazu izravno generira uvjetnu vjerojatnost zavisne varijable $P(Y, \mathbf{X})$. Diskriminativne modele tipično koristimo kad na raspolaganju imamo označene podatke prikladne za nadzirano učenje. U unaprijednom modelu tok informacija je jednosmjeran što znači da (među)rezultati obrade ne mogu biti spojeni unatrag prema izlazu. Osnovni diskriminativni duboki model jest višeslojna unaprijedna neuronska mreža.

Umjetne neuronske mreže

Umjetne neuronske mreže su model strojnog učenja kojeg izražavamo usmjerenim grafom skalarnih procesnih jedinica koje nazivamo umjetnim neuronima. Jedan od važnih ciljeva neuronskih mreža je postavljanje računskog modela biološkog učenja odnosno razumijevanje mehanizama učenja u mozgu živog bića. Iako je vrlo srođno neuronskim mrežama, duboko učenje nema ambiciju modelirati biološke procese, nego proučava učenje kompozicijskih modela od praktičnog značaja koji mogu i ne moraju imati biološku interpretaciju.

Umjetni neuroni tipično provode afinu redukciju ulaznog vektora, što možemo sažeto prikazati izrazom $f(\mathbf{w}^T \mathbf{x} + b)$. Pri tome vektor \mathbf{x} predstavlja ulazne varijable, vektor \mathbf{w} i skalar b predstavljaju slobodne parametre koji se optimiraju postupkom učenja, dok f predstavlja tzv. prijenosnu funkciju umjetnog neurona. Uloga prijenosne funkcije je da u model unese nelinearnost. Ako za f odaberemo funkciju softmax, umjetni neuron će provoditi višerazrednu logističku regresiju. Ako za f odaberemo sigmoidalnu funkciju $\sigma(s) = e^s / (1 + e^s)$, umjetni neuron će provoditi binarnu logističku regresiju. Zbog boljeg učenja dubokih modela, sigmoidu danas istiskuje zglobnica (engl. rectified linear unit, ReLU) $f(s) = \text{ReLU}(s) = \max(0, s)$.

Višeslojne unaprijedne mreže

Neuronska mreža s jednim ulaznim slojem, softmaxom na izlazu, i gubitkom koji maksimizira izglednost parametara ekvivalentna je logističkoj regresiji. Međutim, na ovom kolegiju proučavamo "produbljene" modele koje dobivamo kad između ulaza i izlaza logističke regresije dodamo jednu ili više nelinearnih transformacija. Među njima, posebnu klasu čine unaprijedni modeli u kojima ne postoje povratne veze među neuronima. Takve modele možemo predstaviti acikličkim usmjerjenim grafom gdje čvorovi odgovaraju neuronima, dok lukovi modeliraju povezanost neurona. Poput logističke regresije, unaprijedne duboke modelle najčešće učimo gradijentnim spustom koji optimira izglednost predviđanja modela. Suprotno od logističke regresije, funkcija gubitka dubokih modela nije konveksna, što znači da ne postoji garantija da ćemo naći globalni optimum.

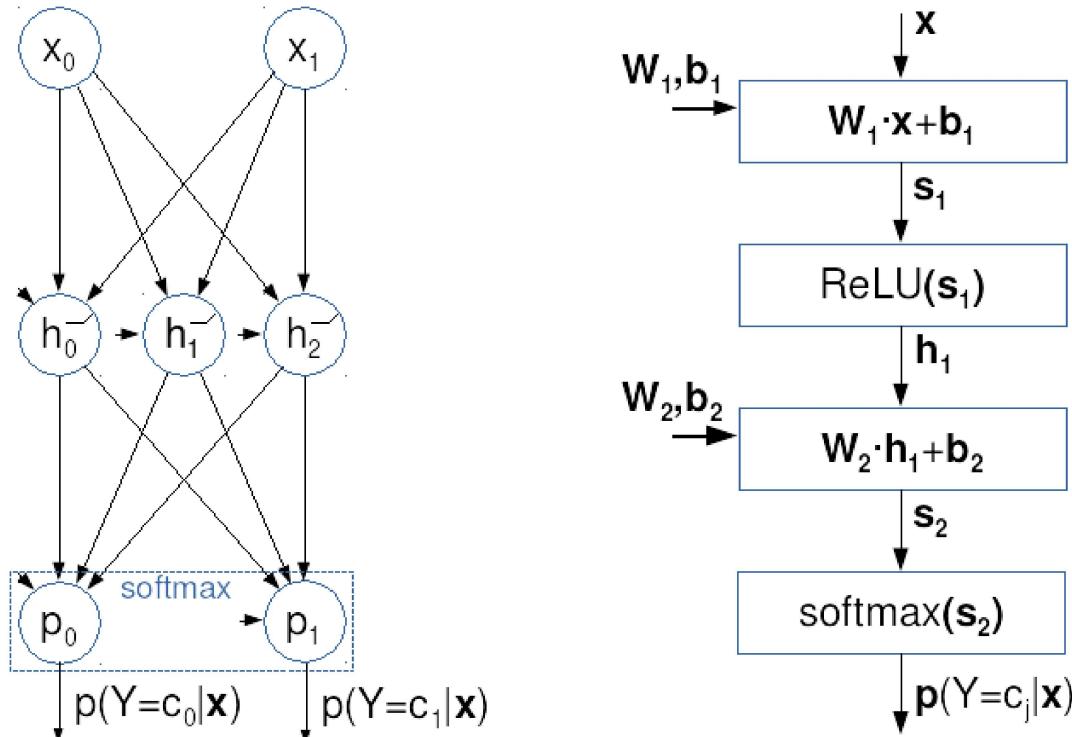
U ovoj vježbi posebno će nam biti zanimljive višeslojne mreže s potpuno povezanim slojevima. U takvim mrežama neurone možemo organizirati u slojeve S_k za koje vrijedi da neuroni sloja k na svojim ulazima primaju sve neurone sloja $k - 1$. Za razliku od logističke regresije, višeslojni modeli mogu modelirati nelinearnu decizijsku granicu, ali po cijeni nekonveksne funkcije cilja.

U posljednje vrijeme popularno je, umjesto pojedinih neurona, čitav sloj promatrati kao kompoziciju linearog i nelinearnog koraka obrade. Ako se dogovorimo da prijenosna funkcija vektorskog operanda odgovara konkatenaciji prijenosnih funkcija komponenata, dolazimo do sljedećeg sažetog zapisa k-tog sloja unaprijedne potpuno povezane mreže sa zglobnom aktivacijom:

$$\mathbf{s}_k = \mathbf{W}_k \cdot \mathbf{h}_{k-1} + \mathbf{b}_k$$

$$\mathbf{h}_k = \text{ReLU}(\mathbf{s}_k)$$

Sljedeća ilustracija prikazuje dva pogleda na istu potpuno povezanu unaprijednu mrežu. Na lijevoj strani je klasični prikaz gdje krugovi odgovaraju neuronima sa skalarnim izlazom i zglobnom prijenosnom funkcijom. Na desnoj strani je vektorizirani računski graf kojeg ćemo koristiti u ovom kolegiju:



Gradijenti u dvoslojnog potpuno povezanom modelu

Razmotrimo sada kako bismo odredili gradijente u prethodno prikazanom primjeru dvoslojnog potpuno povezanog modela. Izrazimo klasifikacijski model vektorskim jednadžbama:

$$\mathbf{s}_1 = \mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1$$

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{s}_1)$$

$$\mathbf{s}_2 = \mathbf{W}_2 \cdot \mathbf{h}_1 + \mathbf{b}_2$$

$$P(Y|\mathbf{x}) = \text{softmax}(\mathbf{s}_2).$$

Naša funkcija gubitka biti će zbroj negativnih log-izglednosti modela u svim podatcima:

$$L(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2 | \mathbf{X}, \mathbf{y}) = \sum_i -\log P(Y = y_i | \mathbf{x}_i)$$

Vidimo da funkcija gubitka u podatku \mathbf{x}_i odgovara kompoziciji većeg broja jednostavnijih funkcija. Gubitak L ovisi o vjerojatnostima P koje ovise o linearnoj klasifikacijskoj mjeri drugog sloja \mathbf{s}_2 koja ovisi o skrivenom sloju \mathbf{h}_1 i parametrima \mathbf{W}_2 i \mathbf{b}_2 . Skriveni sloj \mathbf{h}_1 ovisi o svojoj linearnoj mjeri \mathbf{s}_1 , koja konačno ovisi o parametrima \mathbf{W}_1 i \mathbf{b}_1 te podatcima \mathbf{x} . Stoga gradijente gubitka obzirom na parametre određujemo **ulančavanjem**. Parcijalne derivacije gubitka po j-tim retcima \mathbf{W}_2 i \mathbf{b}_2 biti će vrlo slične onom što smo imali u višerazrednoj logističkoj regresiji. U obzir ćemo uzeti algebarsku strukturu problema, tj. da vrijedi: $\partial s_{2ij} / \partial \mathbf{W}_{2k} = \partial s_{2ij} / \partial \mathbf{b}_{2k} = 0, \forall k \neq j$. Da bismo postigli kompaktniji zapis koristit ćemo matricu aposteriornih vjerojatnosti \mathbf{P} te matricu vektorski kodiranih oznaka \mathbf{Y}' koje smo uveli u nultoj vježbi. Na kraju dobivamo sljedeće izraze:

$$\frac{\partial L_i}{\partial \mathbf{W}_{2j}} = \frac{\partial L_i}{\partial s_{2ij}} \cdot \frac{\partial s_{2ij}}{\partial \mathbf{W}_{2j}} = (P_{ij} - Y'_{ij}) \cdot \mathbf{h}_1^T, \quad ,$$

$$\frac{\partial L_i}{\partial \mathbf{b}_{2j}} = \frac{\partial L_i}{\partial s_{2ij}} \cdot \frac{\partial s_{2ij}}{\partial \mathbf{b}_{2j}} = (P_{ij} - Y'_{ij}).$$

Put do gradijenata po \mathbf{W}_1 i \mathbf{b}_1 nešto je složeniji, jer gradijente treba propagirati preko svih komponenata drugog sloja. Međutim, to propagiranje nije komplikirano jer Jakobijan linearog sloja odgovara matrici težina, dok je Jakobijan zglobnice dijagonalna matrica koja na dijagonali ima nule i jedinice ovisno o predznaku odgovarajuće komponente prvog sloja. Kad konačno dođemo do linearne mjere prvog sloja, možemo iskoristiti izvode koje smo bili dobili u drugom sloju. Ovisnost linearne klasifikacijske mjere drugog sloja o parametrima drugog sloja posve je jednaka ovisnosti linearne mjere prvog sloja o parametrima prvog sloja. Stoga su analitički izrazi parcijalnih derivacija $\partial s_1 / \partial \mathbf{W}_1$ vrlo slični odgovarajućim izrazima u drugom sloju:

$$\frac{\partial L_i}{\partial \mathbf{s}_{1i}} = \frac{\partial L_i}{\partial s_{2i}} \cdot \frac{\partial s_{2i}}{\partial \mathbf{h}_{1i}} \cdot \frac{\partial \mathbf{h}_{1i}}{\partial s_{1i}} = (\mathbf{P}_{i:} - \mathbf{Y}'_{i:}) \cdot \mathbf{W}_2 \cdot \text{diag}(\llbracket s_{1ik} > 0 \rrbracket),$$

$$\frac{\partial L_i}{\partial \mathbf{W}_{1j}} = \frac{\partial L_i}{\partial s_{1ij}} \frac{\partial s_{1ij}}{\partial \mathbf{W}_{1j}} = \frac{\partial L_i}{\partial s_{1ij}} \mathbf{x}_i,$$

$$\frac{\partial L_i}{\partial \mathbf{b}_{1j}} = \frac{\partial L_i}{\partial s_{1ij}} \frac{\partial s_{1ij}}{\partial \mathbf{b}_{1j}} = \frac{\partial L_i}{\partial s_{1ij}}$$

Ovdje valja primijetiti kako naša ambicija nije brzo izračunati *pojedine* gradijente za *pojedine* podatke. Naprotiv, naš cilj je brzo izračunati *sve* gradijente za *sve* podatke oslanjanjem na optimirane biblioteke matrične algebre. S obzirom na to da najveći doprinos brzini možemo ostvariti memorijskim optimizacijama, model trebamo izraziti matričnim operacijama koje djeluju nad svim podatcima. Stoga ćemo, kao i kod logističke regresije gradijente svakog sloja računati odjednom za sve retke parametara i za sve podatke.

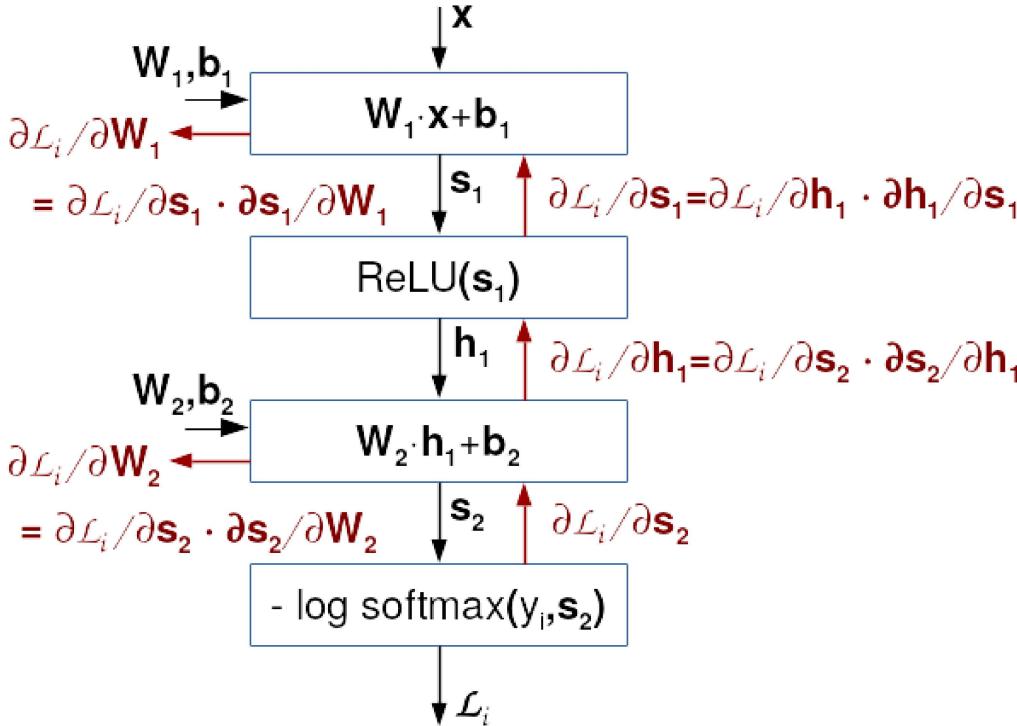
Međutim, za razliku od logističke regresije, u dubokim mrežama moramo donijeti odluku o redoslijedu računanja gradijenata (npr. hoćemo li prije računati $\frac{\partial L_i}{\partial \mathbf{b}_1}$ ili $\frac{\partial L_i}{\partial \mathbf{b}_2}$). Putokaz za rješavanje tog rebusa daje nam algoritam širenja unatrag.

Računanje gradijenata širenjem unatrag

U prikazanim jednadžbama možemo uočiti jednu specifičnost dubokih modela: vidimo da se parcijalna derivacija $\frac{\partial L_i}{\partial s_{2i}}$ javlja u sva četiri gradijenta po parametrima funkcije gubitka. Tu specifičnost možemo iskoristiti kako bismo do željenih gradijenata došli uz minimalni računski napor. Parcijalne derivacije funkcije cilja po čvorovima računskog grafa nećemo morati računati više od jednom ako ih budemo računali *unatrag*, od izlaza

prema ulazu mreže. Taj jednostavni ali vrlo efikasni pristup formalizira algoritam širenja unatrag (engl. backprop).

Postupak širenja pogreške unatrag prikazali smo na sljedećoj slici. Crne strelice prikazuju evaluiranje modela i računanje gubitka u zadanom podatku (tzv. unaprijedni prolaz, engl. forward pass). Crvene strelice prikazuju postupno računanje gradijenata prema algoritmu širenja unatrag (tzv. unatražni prolaz, engl. backward pass).



Sad se čini da su nam poznate sve komponente rješenja našeg problema. Znamo kako računati gradijente s obzirom na pojedine parametre, kao i kojim redoslijedom to obaviti. Međutim, htjeli bismo prije kraja još jednom naglasiti dva netrivijalna detalja. Prvi detalj je petlja po podatcima. Ako želimo uživati prednosti optimiranih biblioteka i izbjegći iteriranje u Pythonu, onda svaki pojedini gradijent trebamo odjednom izračunati za sve podatke. Ako imamo 100000 podataka, prvo ćemo izračunati 100000 redaka matrice

$\mathbf{G}_{\mathbf{s}_2} = [(\frac{\partial L_i}{\partial s_{2i}})_{i=1}^N]$, zatim 100000 redaka matrice $\mathbf{G}_{\mathbf{h}_1} = [(\frac{\partial L_i}{\partial h_{1i}})_{i=1}^N]$, itd. Ovakav pristup je vrlo neobičan za inženjere koji su navikli sve pisati u vlastitom aranžmanu, jer strahovito povećava memoriske zahtjeve postupka. Međutim, tu cijenu moramo platiti, jer u suprotnom naš algoritam ne bismo mogli izraziti optimiranim lego-kockicama za matrične operacije pa bi nam učenje bilo sporije za nekoliko redova veličine.

Drući detalj je računanje gradijenata težina. Ovdje vam preporučamo da umjesto odvojenog računanja gradijenata po retcima težina (kao što sugeriraju gore navedene jednadžbe) koristite pristup kojeg smo u nultoj vježbi pokazali na logističkoj regresiji (isp. odjeljak Od uvodne vježbe). Naime, nije previše teško pokazati da se kompletan matrica gradijenata (koja se u iteraciji gradijentnog spusta naprsto dodaje matrici težina) može izraziti jednostavnim matričnim umnoškom. Gradijente težina u k-tom sloju $\text{grad}(\mathbf{W}_k)$ dobivamo množenjem transponirane matrice gradijenata gubitka po linearnoj mjeri k-tog sloja u svim podatcima $\mathbf{G}_{\mathbf{s}_k}$ s matricom svih ulaza u k-ti sloj \mathbf{H}_{k-1} . Na sličan način računamo i sve gradijente po komponentama pomaka k-tog sloja. Ovdje nam umjesto matričnog množenja treba zbrajanje stupaca matrice $\mathbf{G}_{\mathbf{s}_k}$, a za to možemo koristiti funkciju `np.sum`.

U konkretnom primjeru naše dvoslojne mreže, imali bismo sljedeći redoslijed računanja parcijalnih derivacija funkcije gubitka:

1. gradijenti gubitka obzirom na linearnu mjeru drugog sloja u svim podatcima:

$$\mathbf{G}_{\mathbf{s}_2} = [(\frac{\partial L_i}{\partial s_{2i}})_{i=1}^N] \text{ (matrica dimenzija NxC);}$$

2. gradijenti gubitka obzirom na parametre drugog sloja

- $\text{grad}(\mathbf{W}_2) = [(\frac{\partial L}{\partial \mathbf{W}_{2j}})_{j=1}^C]^\top = \mathbf{G}_{\mathbf{s}_2}^\top \cdot \mathbf{H}_1$ (matrica dimenzija CxH),
- $\text{grad}(\mathbf{b}_2) = [(\frac{\partial L}{\partial b_{2j}})_{j=1}^C]^\top$ (matrica dimenzija Cx1);

3. gradijenti gubitka obzirom na nelinearni izlaz prvog sloja u svim podatcima:

$$\mathbf{G}_{\mathbf{h}_1} = [(\frac{\partial L_i}{\partial \mathbf{h}_{1i}})_{i=1}^N] \text{ (matrica dimenzija NxH);}$$

4. gradijent gubitka obzirom na linearnu mjeru prvog sloja u svim podatcima:

$$\mathbf{G}_{\mathbf{s}_1} = [(\frac{\partial L_i}{\partial \mathbf{s}_{1i}})_{i=1}^N] \text{ (matrica dimenzija NxH);}$$

5. gradijenti gubitka obzirom na parametre prvog sloja:

- $\text{grad}(\mathbf{W}_1) = [(\frac{\partial L}{\partial \mathbf{W}_{1j}})_{j=1}^H]^\top = \mathbf{G}_{\mathbf{s}_1}^\top \cdot \mathbf{X}$ (matrica dimenzija HxD),
- $\text{grad}(\mathbf{b}_1) = [(\frac{\partial L}{\partial b_{1j}})_{j=1}^H]^\top$ (matrica dimenzija Hx1).

Ob. Uvodne napomene o Tensorflowu

Tensorflow je biblioteka otvorenog koda za oblikovanje metoda strojnog učenja s naglaskom na sljedeće ključne mogućnosti:

- automatsko diferenciranje
- izvršavanje na raspodijeljenim i GPU arhitekturama
- bogato klijentsko sučelje prema Pythonu

Iako postoje i drugi alati sa sličnim aspiracijama (theano, torch, caffe i sl.), Tensorflow je zbog čiste arhitekture, bogate biblioteke, efikasne implementacije i sveobuhvatne dokumentacije trenutno jedan od boljih izbora za razne primjene dubokog učenja.

Tensorflow najbolje radi pod Linuxom. Moguće ga je instalirati preko pip-a, a za neke distribucije (npr. arch Linux) dostupni su i binarni paketi. Odnedavna postoji i mogućnost rada na operacijskim sustavima Windows i iOS. Najsvježije informacije o tome možete naći na službenim [stranicama](#).

Kako bi se osigurala maksimalna efikasnost izvođenja na heterogenim arhitekturama, program koji koristi Tensorflow uvijek se sastoji od dvije odvojene faze:

1. oblikovanje računskog grafa,
2. zadavanje računskih upita.

Napomena: u planu je modul `eager` koji će omogućiti relaksiranje tog pravila, ali on još nije ušao u službenu dokumentaciju pa ćemo ga za sada zanemariti.

Ilustrirajmo zadavanje programa u Tensorflowu na jednostavnom primjeru:

```
import tensorflow as tf
# oblikovanje računskog grafa
a = tf.constant(5)
b = tf.constant(8)
x = tf.placeholder(dtype='int32')
c = a + b * x
d = b * x

# fazu zadavanja upita započinjemo
# stvaranjem izvedbenog konteksta:
session = tf.Session()

# zadajemo upit: izračunati c uz x=5
c_val = session.run(c, feed_dict={x: 5})

# ispis rezultata
print(c_val)
```

Prvi dio prikazanog primjera sadrži naredbe koje stvaraju čvorove računskog grafa: a , b , x , c i d . Imenske ovisnosti među izrazima definiraju lukove grafa koji modelira relaciju ovisnosti: čvor c ovisi o čvorovima a , b i x , dok čvor d ovisi o čvorovima b i x . Čvorovi grafa mogu biti konstante (`tf.constant: a, b`), ulazni podatci (`tf.placeholder: x`), tenzorski izrazi (c , d) te parametri postupka (`tf.variable`) i operacije koje ćemo upoznati kasnije. Ulazni podatci, tenzorski izrazi i konstante interno su predstavljeni tipom `tf.Tensor`. Skalarni izrazi se tretiraju kao specijalan slučaj 0-dimenzionalnog tenzorskog izraza.

Računske upite zadajemo pozivima metode `run` izvedbenog konteksta `session`. Prvi argument metode `run` predstavlja listu čvorova koje želimo izračunati. Drugi argument predstavlja rječnik sa vrijednostima ulaznih podataka koje će biti iskorištene tijekom izračuna. Povratna vrijednost metode `run` je n-torka vrijednosti čvorova koji su navedeni u prvom argumentu. Ako želimo izračunati samo jedan čvor, možemo ga navesti i izvan liste, a metoda će tada vratiti njegovu vrijednost izvan n-torke (taj oblik je korišten i u gornjem primjeru).

Primijetite da imena c i c_val referenciraju različite objekte. Ime c referencira čvor grafa u kojem se nalazi operacija $a + b * x$. S druge strane, ime c_value nakon poziva metode `run` referencira broj 45. Izračunavanje tenzorskih izraza i parametara stvara objekt odgovarajućeg numpyjevog podatkovnog tipa (skalara ili višedimenzionalnog polja). Višedimenzionalni ulazni podatci mogu biti zadani ili Numpyjevim objektima ili ugrađenim tipovima Pythona. Prijenos podataka u Tensorflow i iz njega ilustrirat ćemo sljedećim primjerom:

```
import tensorflow as tf
import numpy as np
X = tf.placeholder(tf.float32, [2, 2])
Y = 3 * X + 5
z = Y[0,0]
sess = tf.Session()
Y_val = sess.run(Y, feed_dict={X: [[0,1],[2,3]]})
z_val = sess.run(z, feed_dict={X: np.ones(2,2)})
```

Nakon što primjer utipkamo ili zaliđepimo u interaktivnu ljušku Pythona, rezultate izvođenja možemo provjeriti sljedećim upitim:

```
>>> print(Y_val[0,0], type(Y_val))
5.0 <class 'numpy.ndarray'>

>>> print(z_val, type(z_val))
8.0 <class 'numpy.float32'>
```

Tijekom izvođenja metode `run` Tensorflow prvo određuje relevantni fragment računskog grafa. Ako u našem primjeru zatražimo izračunavanje izraza c , čvorovi a , b , x , i c će biti uvršteni u izračun (tim redoslijedom), dok će se čvor d izostaviti (jer c ne ovisi o d). Nakon toga, relevantni fragment grafa predaje se na izvršavanje izvedbenom modulu odabrane platforme. Ako na računalu postoji slobodni GPU koji korištena verzija Tensorflowa podržava, računske operacije će se izvršiti upravo тамо. Time se omogućava pohranjivanje privremenih vrijednosti na elementu računalnog sustava koji izvršava operacije te posljedično postiže bolja performansa.

0c. Primjena Tensorflowa u strojnog učenju

Kad u Tensorflowu oblikujemo postupak strojnog učenja, računski graf tipično sadržava sljedeće grupe čvorova:

1. parametri (`tf.Variable`) i ulazni podatci (`tf.placeholder`),
2. operacije koje formuliraju model strojnog učenja,
3. operacije koje formuliraju funkciju gubitka,
4. operacije koje formuliraju optimizacijski postupak.

Deklaracije parametara i ulaznih podataka daju osnovne informacije o postupku optimizacije: što optimiramo (parametre) i na temelju čega (podataka). Deklaracije

određuju tip podataka (podrazumijevano `tf.float32`) i dimenzionalnost (ako se radi o vektoru, matici ili tenzoru). Informacija o tipu podatka se koristi prilikom prevođenja računskog grafa u optimirani izvršni kod za odabranu arhitekturu (dakle, Tensorflow interno koristi staticko tipiziranje). Kako bi se omogućio rad sa skupovima podataka različite brojnosti, dimenzije čvorova ulaznih podataka (`placeholder`) mogu biti zadane ključnom rječju `None`, kao u sljedećem primjeru.

```
X = tf.placeholder(tf.float32, [None, 5])
Y = 3 * X + 5
```

U prikazanom primjeru deklarirali smo matricu nepoznatog broja 5-dimenzionalnih ulaznih podataka. Dimenzije ulaznog tenzora `X` i međurezultata `Y` postati će poznate tek u trenutku zadavanja računskog upita kad će konkretna vrijednost matrice `X` biti zadana argumentom `feed_dict`. Nakon izvođenja sljedećeg koda, `Y_val` će pokazivati na numpyjevu matricu dimenzija 3x5 čiji će svi elementi biti jednaki 8.

```
sess = tf.Session()
Y_val = sess.run(Y, feed_dict={X: np.ones((3,5))})
```

Tensorflow razlikuje statičke i dinamičke dimenzije tenzora. Statičke dimenzije tenzora određuju se na temelju deklaracija ulaznih podataka, a možemo ih dobiti pozivom metode `get_shape`. Ako neke dimenzije nisu zadane deklaracijom, odgovarajuća statička dimenzija biti će `None`. Tako će sljedeći kod ispisati `TensorShape([Dimension(None), Dimension(5)])`:

```
print(Y.get_shape())
```

Ako je dinamička dimenzionalnost ulaznog polja važna za naš graf, do nje možemo doći pozivom funkcije `tf.shape`. Tako će sljedeći odsječak ispisati `[3, 5]`.

```
print(sess.run(tf.shape(Y), feed_dict={X: np.ones((3,5))}))
```

Model strojnog učenja zadajemo pythonskim kodom koji formalizira vezu između ulaznih podataka i predikcije modela. Na sličan način zadajemo i funkciju gubitka koja formalizira "dobrotu" predikcije modela u odnosu na zadane željene rezultate. Često ćemo se izražavati uz pomoć elemenata bogatog **klijentskog sučelja** Tensorflowa. Tensorflow podržava složene operacije nad vektorima, maticama i tenzorima (slično kao i numpy) što pogoduje prevođenju u efikasan izvršni kod za grafičke procesore.

Optimizacijski postupak definira na koji način funkcija gubitka utječe na napredovanje parametara modela. Najjednostavniji postupak je gradijentni spust kojeg implementira razred `tf.train.GradientDescentOptimizer`. Metoda `minimize` tog razreda stvara čvor grafa (operaciju) koja provodi jednu iteraciju gradijentnog spusta: računa gradijente funkcije gubitka obzirom na parametre, te uvećava parametre u smjeru negativnog gradijenta.

Tipična implementacija algoritma strojnog učenja sastoji se od sljedećih koraka:

1. definicija računskog grafa: to smo objasnili gore;
2. incijalizacija izvedbenog konteksta: stvaramo objekt tipa `tf.Session` koji pamti tekuće vrijednosti parametara te provodimo incijalizaciju parametara zadavanjem operacije koju vraća funkcija `tf.initialize_all_variables`;
3. učenje: iterativno izračunavamo čvor grafa kojeg smo stvorili metodom `minimize` odabranog gradijentnog postupka; analitičko deriviranje funkcije gubitka Tensorflow provodi automatski :-)

U mnogim postupcima dubokog učenja, parametre težina ćemo incijalizirati slučajnim vrijednostima. Kako biste tijekom razvoja postupka osigurali ponovljivost rezultata, generator slučajnih brojeva možete incijalizirati na konstantu:

```
tf.set_random_seed(100)
```

1. Generiranje linearno nerazdvojivih podataka

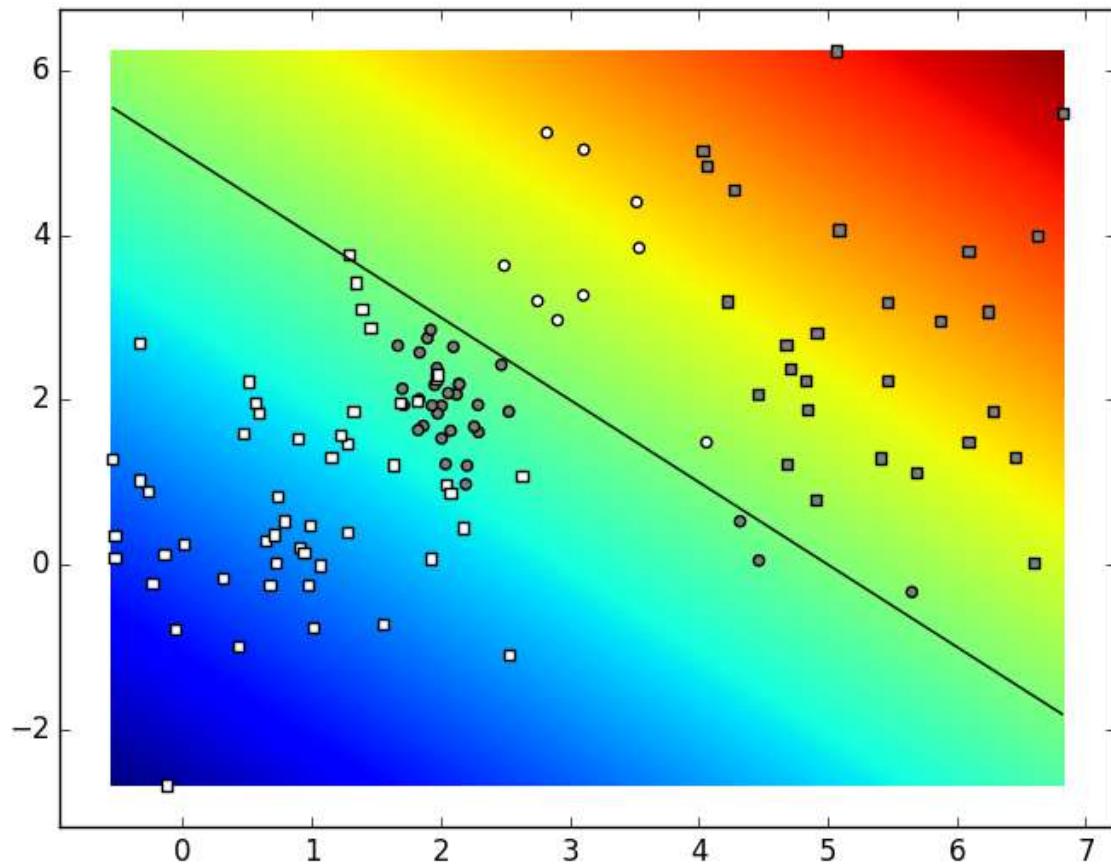
U dosadašnjim eksperimentima logistička regresija je postizala iznimno dobre klasifikacijske rezultate. To nije nikakvo čudo jer se pokazuje da aposteriorna vjerojatnost razreda podataka generiranih Gaussovim razdiobama s dijeljenom kovarijancom odgovara upravo sigmoidi afino transformiranih podataka. Istina, u našim smo eksperimentima mrvicu odstupili od teoretskih pretpostavki (naši razredi su imali različite kovarijance), ali rezultati pokazuju da to odstupanje nije bilo presudno. Sada ćemo situaciju malo otežati na način da pozitivne i negativne podatke generiramo nešto složenijim generativnim modelom.

Upute:

- Napišite potprogram `sample_gmm_2d(K, c, N)` koja stvara K slučajnih bivarijatnih Gaussovih razdioba, te iz svake od njih uzorkuje N podataka. Za razliku od funkcije `sample_gauss_2d` ovdje svakoj bivarijatnoj razdiobi G_i trebamo pridružiti razred c_i kojeg slučajno biramo iz skupa $\{0, 1, \dots, C-1\}$. Na taj način dobivamo podatkovne razrede generirane mješavinama slučajno odabralih Gaussovih razdioba. Kao i ranije, funkcija treba vratiti matricu x čiji retci odgovaraju uzorkovanim podatcima te matricu y čiji jedini stupac sadrži indeks razdiobe iz koje je uzorkovan odgovarajući podatak.
- U izvedbi potprograma potrebno je prvo instancirati slučajne distribucije i svakoj od njih dodijeliti slučajan razred od 0 do $C - 1$. Zatim je potrebno iz svake distribucije uzorkovati traženi broj podataka. Svi podatci uzorkovani iz iste distribucije trebaju dobiti indeks razreda koji je dodijeljen toj distribuciji.
- Potprogram treba vratiti sljedeće podatke:

```
...
x ... podatci u matrici [K·N x 2 ]
y_ ... indeksi razreda podataka [K·N]
...
```

Izvedite potprogram `sample_gmm_2d` te ga ispitajte uz pomoć prethodno razvijenih potprograma za crtanje (`graph_surface` i `graph_data`). Ovisno o parametrima i stanju generatora slučajnih brojeva, vaš rezultat mogao bi izgledati kao na sljedećoj slici. Naši parametri bili su $K=4$, $C=2$, $N=30$.

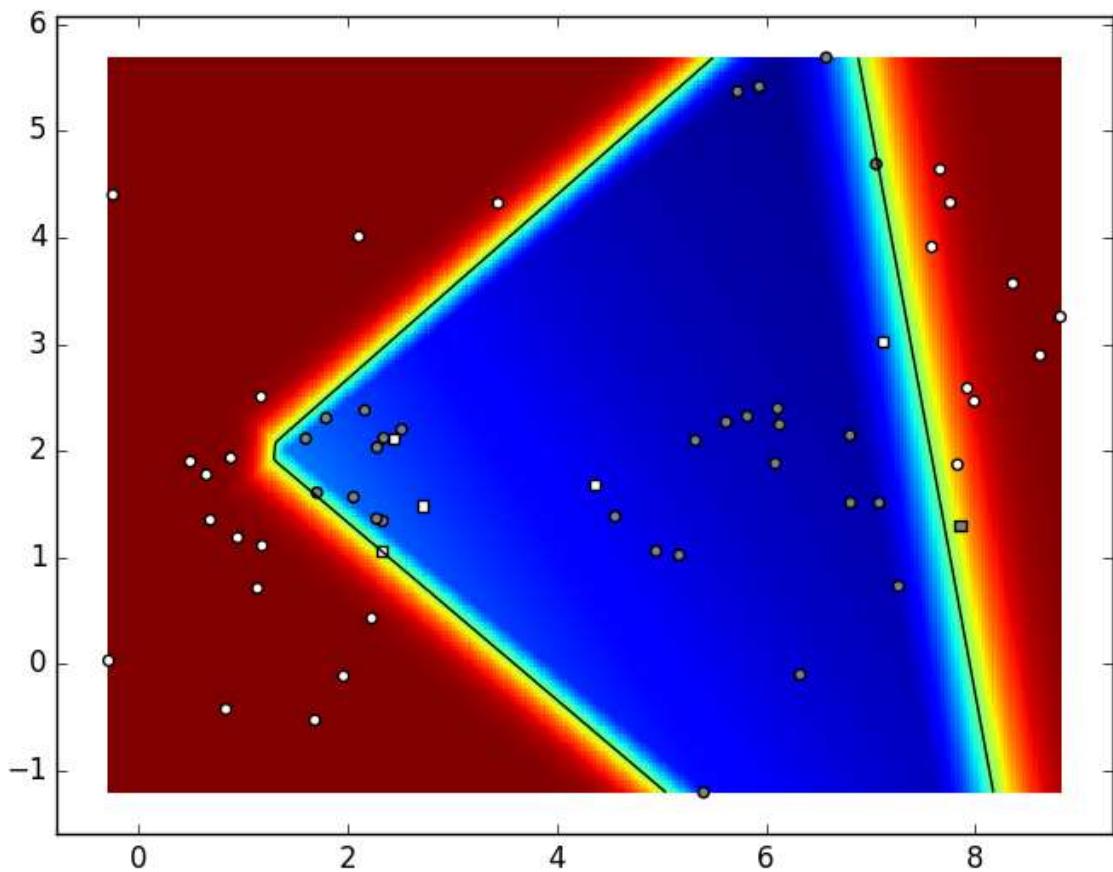


Ako je rezultat izvođenja prihvatljiv, pohranite kod u datoteku `data.py`.

2. Višeslojna klasifikacija u Pythonu

Oblikujte i izvedite modul `fcann2` za rad s probabilističkim klasifikacijskim modelom s jednim skrivenim slojem. Neka organizacija vašeg koda bude sukladna organizaciji modula `logreg` iz prethodne vježbe. Napišite metode `fcann2_train`, `fcann2_classify`. Isprobajte njihov rad na umjetnom skupu 2D podataka dvaju razreda dobivenih iz Gaussove mješavine od 6 komponenata.

Ovisno o parametrima i stanju generatora slučajnih brojeva, vaš rezultat mogao bi izgledati kao na sljedećoj slici. Naši parametri bili su: $K=6$, $C=2$, $N=10$, $\text{param_niter}=1e5$, $\text{param_delta}=0.05$, $\text{param_lambda}=1e-3$ (koeficijent regularizacije), dimenzija skrivenog sloja: 5.



Ako je rezultat prihvatljiv, pohranite kod u datoteku fcann2.py.

3. Linearna regresija u Tensorflowu

Tipičnu strukturu implementacije algoritma strojnog učenja u Tensorflowu prikazat ćeemo na potpunom primjeru optimizacijskog postupka za određivanje parametara pravca $y = a * x + b$ koji prolazi kroz točke (1,3) i (3,5).

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

## 1. definicija računskog grafa
# podatci i parametri
X = tf.placeholder(tf.float32, [None])
Y_ = tf.placeholder(tf.float32, [None])
a = tf.Variable(0.0)
b = tf.Variable(0.0)

# afini regresijski model
Y = a * X + b

# kvadratni gubitak
loss = (Y-Y_)**2

# optimizacijski postupak: gradijentni spust
trainer = tf.train.GradientDescentOptimizer(0.1)
train_op = trainer.minimize(loss)

## 2. inicializacija parametara
sess = tf.Session()
sess.run(tf.initialize_all_variables())

## 3. učenje
# neka igre počnu!
for i in range(100):
```

```
val_loss, _, val_a, val_b = sess.run([loss, train_op, a, b],
    feed_dict={X: [1,2], Y: [3,5]})
print(i, val_loss, val_a, val_b)
```

Zadatci:

- Analizirajte prikazani program te provjerite ispravnost izvođenja.
- Bi li ovaj program "znao" odrediti pravac definiran proizvoljnim brojem točaka? Ako da, isprobajte to.
- Izrazite korak optimizacije postupka pozivima metoda `compute_gradients` i `apply_gradients`.
- Ispišite vrijednosti gradijenata tijekom napredovanja postupka tako da u poziv metode `run` uključite i gradiente koje vraća funkcija `compute_gradients`. Pripazite, funkcija `compute_gradients` vraća listu parova (gradijent, varijabla), dok metoda `run` izvedbenog konteksta u jedinom pozicijskom argumentu prima listu *pojedinačnih* čvorova računskog grafa poput npr. `loss, train_op, a` ili `b`.
- Odredite analitičke izraze za gradiente funkcije gubitka po parametrima `a` i `b`. Dodajte čvorove grafa koji eksplicitno računaju te gradijente i uvrstite ih u računski upit zadan metodom `run`. Ispišite vrijednosti gradijenata i uvjerite se da odgovaraju onima koje automatski određuje Tensorflow.
- Pokažite da se vrijednosti gradijenata mogu ispisati i primjenom funkcije `tf.Print`.

Ako je rezultat prihvatljiv, pohranite kod u datoteku `tf_linreg.py`.

4. Logistička regresija u Tensorflowu

U ovom zadatku ćemo postupak logističke regresije izvesti uz pomoć Tensorflowa. Dobiveni kod će biti oko dvostruko kraći od "ručnog rada" koji je bio predmet nulte vježbe. Glavne prednosti Tensorflowa su u tome što ne moramo izvoditi gradijente te što dobiveni program bez ikakvih izmjena može izvršavati na moćnim grafičkim karticama. Te prednosti će se pokazati presudnima kod velikih modela sa stotinama milijuna slobodnih parametara (kod malih modela procesori opće namjene mogu biti brži od grafičkih procesora zbog dugotrajnog prebacivanja podataka).

U uvodnim primjerima smo vidjeli da program koji koristi Tensorflow za mnoge veličine treba imati dva imena (npr. `loss` i `val_loss` te `a` i `val_a`). Prvo ime nam treba za referenciranje čvora računskog grafa, dok drugo povezujemo sa stvarnom vrijednošću koju to ime poprima za konkretnе podatke. Kako bismo izbjegli potrebu za dvostrukim imenima zgodno je algoritam izraziti razredom čiji podatkovni elementi sadrže čvorove grafa dok klijentski kod ista imena koristi za pohranjivanje konkretnih vrijednosti koje ti čvorovi poprimaju u pojedinim upitim. Stoga bi razred za logističku regresiju nad Tensorflowom mogao imati sljedeću strukturu.

```
class TFLogreg:
    def __init__(self, D, C, param_delta=0.5):
        """Arguments:
            - D: dimensions of each datapoint
            - C: number of classes
            - param_delta: training step
        """

        # definicija podataka i parametara:
        # definirati self.X, self.Yoh_, self.W, self.b
        # ...

        # formulacija modela: izračunati self.probs
        #   koristiti: tf.matmul, tf.nn.softmax
        # ...

        # formulacija gubitka: self.loss
        #   koristiti: tf.log, tf.reduce_sum, tf.reduce_mean
        # ...

        # formulacija operacije učenja: self.train_step
        #   koristiti: tf.train.GradientDescentOptimizer,
        #             tf.train.GradientDescentOptimizer.minimize
```

```

# ...

# instanciranje izvedbenog konteksta: self.session
#   koristiti: tf.Session
# ...

def train(self, X, Yoh_, param_niter):
    """Arguments:
        - X: actual datapoints [NxD]
        - Yoh_: one-hot encoded labels [NxC]
        - param_niter: number of iterations
    """
    # incijalizacija parametara
    #   koristiti: tf.initialize_all_variables
    # ...

    # optimizacijska petlja
    #   koristiti: tf.Session.run
    # ...

def eval(self, X):
    """Arguments:
        - X: actual datapoints [NxD]
        Returns: predicted class probabilites [NxC]
    """
    #   koristiti: tf.Session.run

```

Primijetite da za razliku od prethodne vježbe točne oznake razreda podataka za učenje sada nazivamo Y_{oh} umjesto $y_$. Razlog tome je što unakrsnu entropiju u Tensorflowu lakše izražavamo kad su oznake smještene u matici gdje retci odgovaraju podatcima, a stupci razredima (tzv. one-hot notacija). Ako podatak x_i odgovara razredu c_j , onda vrijedi $Y_{oh}[i,j]=1$ te $Y_{oh}[i,k]=0$ za $k \neq j$ ("one hot"). Podsjetimo se, tako organizirane oznake razreda u ranijim matematičkim izrazima nazivali smo matricom vektorski kodiranih oznaka \mathbf{Y}' .

Struktura ispitnog programa bila bi vrlo slična ispitnim programima iz prethodne vježbe:

```

if __name__ == "__main__":
    # incijaliziraj generatore slučajnih brojeva
    np.random.seed(100)
    tf.set_random_seed(100)

    # instanciraj podatke X i labele Yoh_
    # izgradi graf:
    tflr = TFLogreg(X.shape[1], Yoh_.shape[1], 0.5)

    # nauči parametre:
    tflr.train(X, Yoh_, 1000)

    # dohvati vjerojatnosti na skupu za učenje
    probs = tflr.eval(X)

    # ispiši performansu (preciznost i odziv po razredima)

    # iscrtaj rezultate, decizijsku plohu

```

Zadatci:

- Dopunite izvedbu razreda `TFLogreg` te provjerite postiže li vaš program iste rezultate kao i odgovarajući program iz nulte vježbe za slučajevе dva i tri razreda podataka.
- Dodajte regularizaciju na način da gubitak formulirate kao zbroj unakrsne entropije i L2 norme vektorizirane matrice težina pomnožene hiperparametrom `param_lambda`. Ispitajte utjecaj regularizacije na oblik decizijske plohe.
- Eksperimentirajte s različitim vrijednostima hiperparametara. Pronađite kombinacije hiperparametara za koje vaš program ne uspijeva pronaći zadovoljavajuće rješenje i pokušajte objasniti što se događa.

Ako je rezultat izvođenja prihvatljiv, pohranite kod u datoteku `tf_logreg.py`.

5. Konfigurabilni duboki modeli u Tensorflowu

Naš sljedeći zadatak je proširiti izvedbu logističke regresije na način da omogućimo jednostavno zadavanje potpuno povezanih modela proizvoljne dubine. Nazovimo naš novi razred `TFDeep`. Neka sučelje tog razreda bude posve identično sučelju razreda `TFLogreg`, osim što ćemo u konstruktoru umjesto dimenzionalnosti podataka i broja razreda zadati listu cijelih brojeva koji će određivati broj neurona u svakom sloju. Nulti element te liste definira dimenzionalnost podataka, dok njen posljednji element (na rednom broju $n-1$) odgovara broju razreda. Elementi konfiguracije na indeksima od 1 do $n-2$ (ako postoje) sadržavaju brojeve neurona u skrivenim slojevima. Tako konfiguracija `[2, 3]` odgovara logističkoj regresiji dvodimenzionalnih podataka u tri razreda. Konfiguracija `[2, 5, 2]` odgovara modelu s jednim skrivenim slojem h koji se sastoji od 5 neurona:

```
h = f(X * W_1 + b_1)
probs = softmax(h * W_2 + b_2)
```

U posljednjem primjeru dimenzije čvorova grafa trebaju biti kako slijedi (upitnici označavaju nepoznatu brojnost skupa podataka na kojem primijenjujemo model):

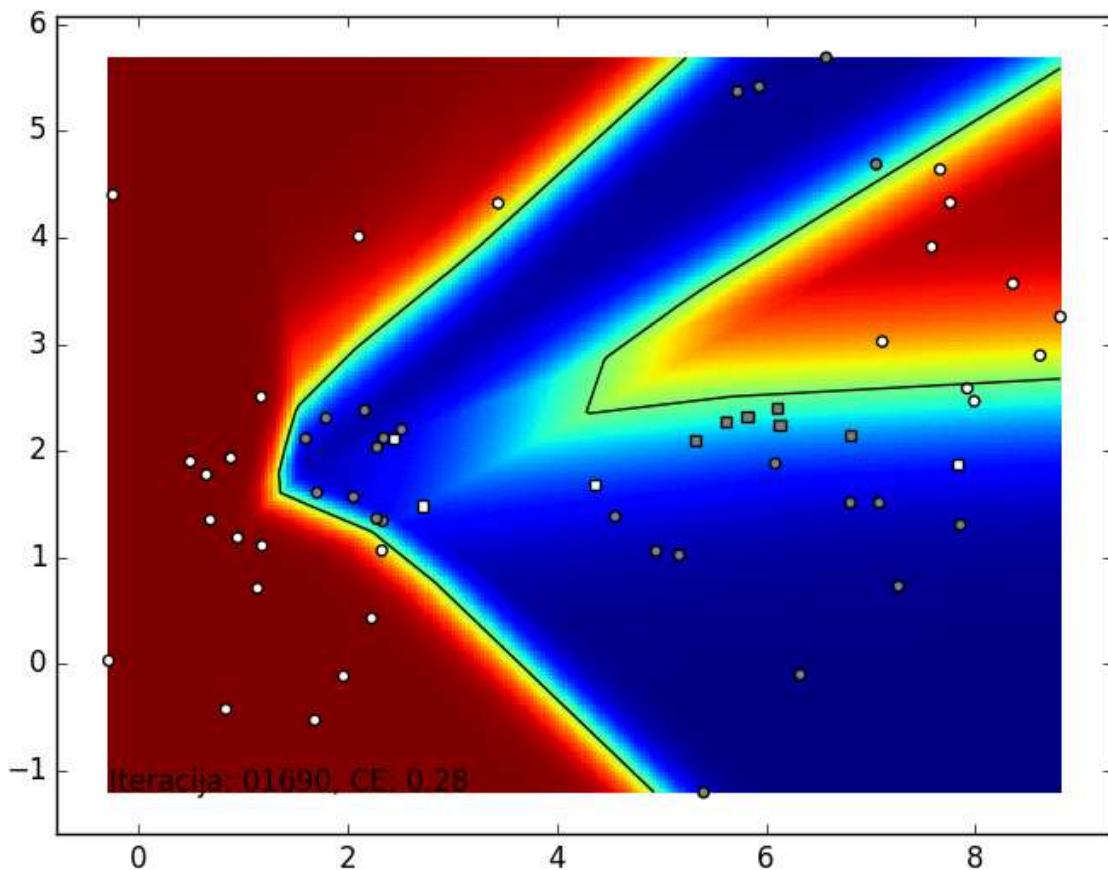
```
X      ... [?, 2]
W_1    ... [2, 5]
b_1    ... [1, 5]
h_1    ... [?, 5]
W_2    ... [5, 3]
b_2    ... [1, 3]
probs ... [?, 3]
```

Implementacija razreda `TFDeep` biti će vrlo slična implementaciji razreda `TFLogreg`. Najviše posla ćemo imati u konstruktoru jer moramo izraziti konstrukciju grafa u ovisnosti konfiguracijskoj listi. S obzirom na to da broj slojeva može biti različit, matrice težina i vektore pomaka trebat će smjestiti u liste (nazovimo ih `self.W` i `self.b`). Jednako vrijedi i za matrice skrivenih slojeva koje možemo pohraniti u listu `self.h`. Definiciju dimenzija parametara možemo provoditi iz petlje po indeksima konfiguracijske liste. Radi veće jasnoće izvornog koda, predlažemo vam da formulaciju modela izvedete u zasebnoj petlji. Nelinearnost u skrivenim slojevima možete izraziti uz pomoć funkcija TensorFlowa `tf.ReLU`, `tf.sigmoid` odnosno `tf.tanh`.

Zadatci:

- Izvedite razred `TFDeep` te isprobajte konfiguraciju `[2, 3]` na istim podatcima kao i u prethodnom zadatku (ispitni program će vam biti vrlo sličan). Provjerite da su rezultati isti kao i ranije.
- Ako pri definiranju parametara pozivom funkcije `tf.Variable` niste zadali simbolička imena, napravite to sada (po potrebi proučite [dokumentaciju](#)). Napišite metodu `count_params` koja ispisuje simbolička imena svih parametara koji se mogu učiti obilaženjem povratne vrijednosti funkcije `tf.trainable_variables`. Ispišite i ukupan broj svih komponenata parametara (npr. za konfiguraciju `[2, 3]` rezultat bi trebao biti 9).
- Isprobajte vaš kod na podatcima dobivenim pozivima `data.sample_gmm_2d(4, 2, 40)` i `data.sample_gmm_2d(6, 2, 10)`, za konfiguracije `[2, 2]`, `[2, 10, 2]` i `[2, 10, 10, 2]`. Ispišite točnost, odziv, preciznost i prosječnu preciznost te grafički prikažite rezultate klasifikacije i izgled decizijske plohe. Ako ne dođe do konvergencije, obratite pažnju na vrijednosti hiperparametara.
- Usporedite rezultate s onim što se zbiva kad za prijenosnu funkciju postavite sigmoidu. Sigmoida bi za ovakve male probleme zbog neprekidnosti trebala postići bolje rezultate od zglobnice. Glavna prednost zglobnice je u tome što nema zasićenje pa kod dubljih modela gradjenti teže nestaju.

Ovisno o parametrima i stanju generatora slučajnih brojeva, vaš rezultat mogao bi izgledati kao na sljedećoj animaciji (naši parametri bili su: $\kappa=6$, $C=2$, $N=10$, $\text{param_niter}=1\text{e}4$, $\text{param_delta}=0.1$, $\text{param_lambda}=1\text{e}-4$ (koeficijent regularizacije), $\text{config}=[2, 10, 10, 2]$, `ReLU`).



Ako je rezultat izvođenja prihvatljiv, vaš kod pohranite u datoteku `tf_deep.py`.

6. Usporedba s jezgrenim SVM-om

Podsjetite se na svojstva jezgrenog SVM-a (model, gubitak, optimizacija) te pročitajte dokumentaciju modula `svm` biblioteke `scikit-learn`. Oblikujte razred `KSVMWrap` kao tanki omotač oko modula `sklearn.svm` kojeg ćemo moći primijeniti na našim dvodimenzionalnim podatcima. S obzirom na to da će konstrukcija omotača biti jednostavna, učenje možemo provesti iz konstruktora, dok predikciju, rezultate i dohvati potpornih vektora možemo izvesti u metodama. Neka sučelje razreda bude kako slijedi:

```
...
Metode:
__init__(self, X, Y_, param_svm_c=1, param_svm_gamma='auto'):
    Konstruira omotač i uči RBF SVM klasifikator
    X,Y_:          podatci i točni indeksi razreda
    param_svm_c:   relativni značaj podatkovne cijene
    param_svm_gamma: širina RBF jezgre

predict(self, X)
    Predviđa i vraća indekse razreda podataka X

get_scores(self, X):
    Vraća klasifikacijske mjerne podataka X

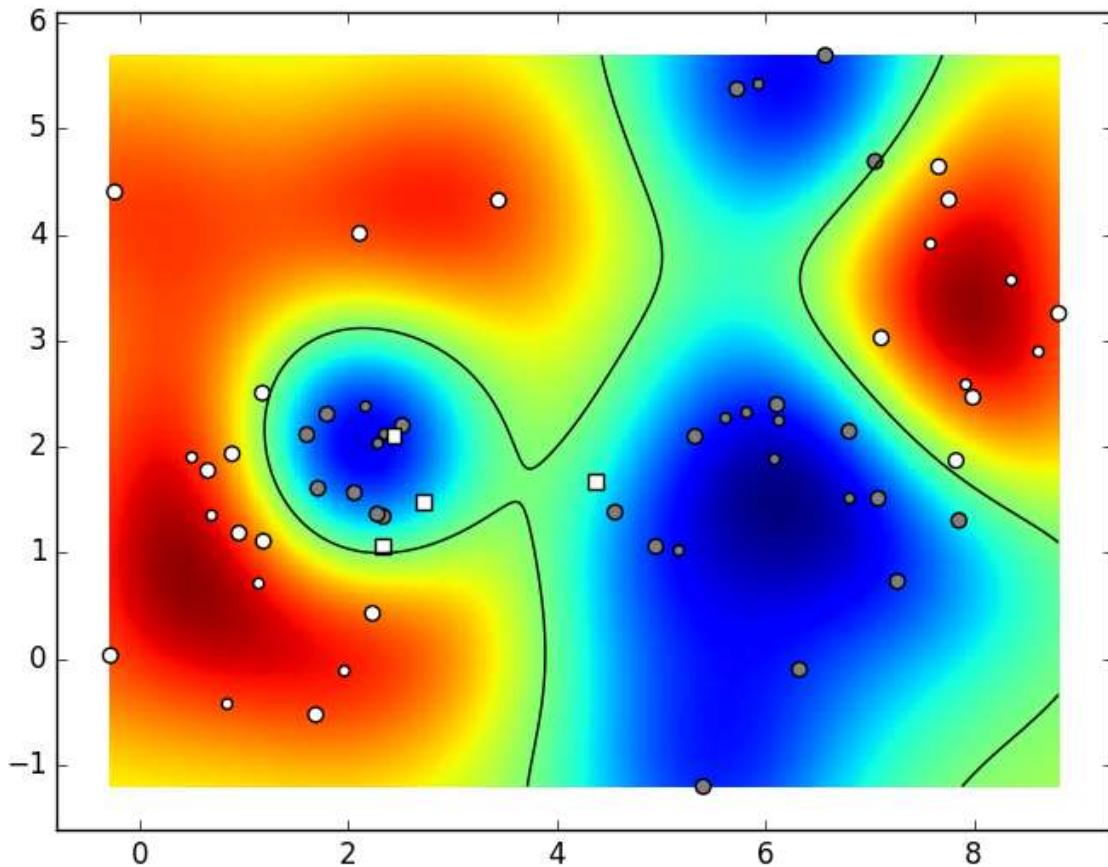
    suport
        Indeksi podataka koji su odabrani za potporne vektore
...

```

Zadatci:

- Modificirajte funkciju `data.graph.data` na način da joj dodate argument `special`. Argument `special` zadaje listu indeksa podataka koje prilikom iscrtavanja treba posebno naglasiti udvostručavanjem veličine njihovih simbola.

- Isprobajte vaš razred na podacima dvaju razreda uzorkovanih iz mješavina Gaussovi distribucija. Kao i obično, ispišite pokazatelje performanse (točnost, odziv, preciznost, prosječnu preciznost).
- Usporedite performansu modela koje implementiraju razredi `TFDeep` i `KSVMLwrap` na većem broju slučajnih skupova podataka. Koje su prednosti i nedostatci njihovih funkcija gubitka? Koji od dvaju postupaka daje bolju garantiranu performansu? Koji od postupaka može primiti veći broj parametara? Koji bi od postupaka bio prikladniji za 2D podatke uzorkovane iz mješavine Gaussovi distribucija?
- IsCRTajte decizijsku plohu i rezultate klasifikacije RBF SVM-a. Iskoristite argument `special` funkcije `data.graph.data` da u prikazu podataka posebno istaknete potporne vektore. Ovisno o parametrima i stanju generatora slučajnih brojeva, vaš rezultat mogao bi izgledati kao na sljedećoj animaciji (naši parametri bili su: $k=6$, $C=2$, $N=10$, `param_svm_c=1`, `param_svm_gamma='auto'`).



Ako je rezultat izvođenja prihvatljiv, pohranite kod u datoteku `ksvm_wrap.py`.

7. Studija slučaja: MNIST

U dosadašnjim vježbama naučene modele nismo evaluirali na nezavisnom skupu za testiranje. Takvi eksperimenti ne bi nužno otkrili generalizacijski potencijal algoritama, jer se generativni modeli stvarnih podataka ne moraju moći opisati Gaussovim razdiobama. Zato ćemo se u ovoj vježbi posvetiti generalizacijskoj performansi na stvarnom skupu podataka MNIST.

MNIST predstavlja skup slika rukom pisanih znamenki od 0 do 9. Svaka znamenka predstavljena je slikom dimenzija 28x28 piksela. MNIST sadrži 50000 slika u skupu za učenje i 10000 slika u skupu za testiranje. MNIST možemo jednostavno učitati sljedećim kodom:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

```
tf.app.flags.DEFINE_string('data_dir',
    '/tmp/data/', 'Directory for storing data')
mnist = input_data.read_data_sets(
    tf.app.flags.FLAGS.data_dir, one_hot=True)
```

Sada su skupovi slika i indeksi razreda predstavljene numpyjevim matricama `train.images`, `train.labels`, `test.images` i `test.labels` u atributima objekta `mnist`. Do dimenzija podataka možemo doći jednostavnim propitivanjem oblika tih matrica.

```
N=mnist.train.images.shape[0]
D=mnist.train.images.shape[1]
C=mnist.train.labels.shape[1]
```

Pojedinačne slike odgovaraju retcima matrica. Ako ih želimo prikazati, retke je prvo potrebno preoblikovati na originalne dimenzije 28x28 te dobivene matrice poslati funkciji `plt.imshow` uz argumente `cmap = plt.get_cmap('gray')`, `vmin = 0` te `vmax = 1`.

Zadatci:

- Za model konfiguracije [784,10] iscrtajte i komentirajte matrice težina za svaku pojedinu znameku.
- Naučite duboke modele s konfiguracijama [784,10], [784,100,10], [784,100,100,10] i [784,100,100,100,10] te usporedite njihove performanse i matrice zabune na skupovima za učenje i testiranje. Ako nemate funkcionalan GPU ne morate provoditi eksperimente s posljednje dvije konfiguracije. Obratite pažnju na to da će dublji modeli bolje konvergirati s više iteracija s manjim korakom. Za najuspješniji model iscrtajte podatke koji najviše doprinose funkciji gubitka.
- Proučite utjecaj regularizacije na performansu dubokih modela na skupovima za učenje i testiranje.
- Slučajno izdvojite 1/5 podataka iz skupa za učenje u skup za validaciju. Tijekom treniranja evaluirajte validacijsku performansu nakon završetka petlje po grupama podataka te na kraju vratite model s najboljom validacijskom performansom (engl. early stopping). Procijenite postignuti utjecaj na konačnu vrijednost funkcije cilja i generalizacijsku performansu.
- Implementirajte stohastički gradijentni spust odnosno postupak učenja po mini-grupama. Prije svake epohe izmiješajte podatke, zatim ih podijelite u n grupa (engl. mini-batch) i onda provedite korak učenja za svaku grupu posebno. Vaš kod pohranite u metodi `train_mb` razreda `TFDeep`. Procijenite utjecaj na kvalitetu konvergencije i postignutu performansu za najuspješniju konfiguraciju iz prethodnog zadatka.
- Promijenite optimizator u `tf.train.AdamOptimizer` s fiksnim korakom učenja `1e-4`. Procijenite utjecaj te promjene na kvalitetu konvergencije i postignutu performansu.
- Isprobajte ADAM s varijabilnim korakom učenja. U izvedbi se pomognite funkcijom `tf.train.exponential_decay`. Neka početni korak učenja bude isti kao i ranije, a ostale parametre postavite na `decay_rate=1-1e-4` i `decay_steps=1`.
- Naučite linearni i jezgreni SVM uz pomoć modula `sklearn.svm`. Koristite podrazumijevano `one vs one` proširenje SVM-a za klasificiranje podataka u više razreda. Pri eksperimentiranju budite strpljivi jer bi učenje i evaluacija mogli trajati više od pola sata. Usporedite dobivenu performansu s performansom dubokih modela.

Ako je rezultat izvođenja prihvatljiv, pohranite kod u datoteku `mnist_shootout.py`.

8. Normalizacija po podatcima (bonus)

Proučite [postupak](#) normalizacije po podatcima (engl. batch normalization) za potpuno povezane modele. Proširite duboki klasifikator kodom koji normalizira linearni dio svakog skrivenog sloja tako da za tekuću grupu ima nultu sredinu i jediničnu varijancu. Pripazite na to da parametre normalizacije valja mijenjati samo prilikom [učenja](#). Usporedite dobivenu performansu s performansom osnovnih dubokih modela.

Ove stranice su rezultat rada na istraživačkom projektu [MULTICLOUD](#) (I-2433-2014) Hrvatske zaklade za znanost.

Stranice su izrađene [vi-jem](#) i [geditom](#).

Posljednja promjena: Friday, 20-Oct-2017 18:11:01 CEST

Svi komentari su dobrodošli: sinisa.segvic@fer.hr

[Povratak](#)