Randomizirana metoda optimizacije ADAM

Marić, Igor

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Igor Marić

RANDOMIZIRANA METODA OPTIMIZACIJE ADAM

Diplomski rad

Zagreb, travanj, 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

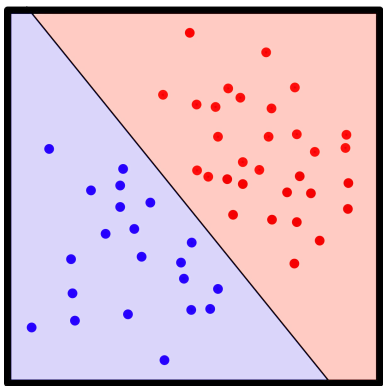
1. _____
2. _____
3. _____

Zahvaljujem se mentoru prof. dr. sc. Luki Grubišiću na brojnim sugestijama, pomoći i usmjeravanju tijekom pisanja rada. Hvala svim prijateljima i kolegama zbog kojih će mi studentski život ostati u lijepom sjećanju. Veliko hvala roditeljima i bratu koji su uvijek bili bezuvjetna podrška i bez kojih ne bih ispunio ovaj cilj.

Sadržaj

Uvod

Tema ovog diplomskog rada su stohastičke gradijentne metode u kontekstu minimizacije separabilnih funkcija. Kažemo da je funkcija separabilna (jednostavnosti radi, uzmimo 2 nezavisne varijable x i y) ako ju je moguće prikazati kao produkt ili sumu dvaju funkcija, f_1 i f_2 koje su ovisne samo o jednoj varijabli. Matematičkim jezikom, $f(x, y) = f_1(x)f_2(y)$, $f(x, y) = f_1(x) + f_2(y)$ respektivno. Primjer funkcije koja je separabilna u obliku sume je empirijska funkcija rizika koja se koristi kao procjena greške kojom dani regresijski model reprezentira skup ulazno izlaznih parova. Funkcija je separabilna budući da je dana kao suma grešaka aproksimacija pojedinog para ulazno izlaznih opažanja.



Slika 0.1: Linearna separabilnost; Funkciju koja točkama u ravnini pridružuje plavu ili crvenu boju reprezentiramo potpornim vektorima $x \mapsto \text{sign}(\omega^T x)$, za sign funkciju predznaka i dani vektor prametara θ. Izvor:[18]

Dotične metode su naširoko uporabljive u domeni optimizacije i dubokog učenja.

Strojno učenje je grana umjetne inteligencije (artificial inteligence – AI) i računarske znanosti. Sam izraz pripisujemo Arthuru Samuelu, zaposleniku IBM-a koji je razvio AI računala za povlačenje poteza u igrama na ploči, konkretno u igri dama. Više se može pročitati u [5]. Vrtoglavi tehnološki napredak računala u vidu porasta raspoložive memo-

rije za pohranu podataka i procesuiranje računala svode prethodni problem na trivijalnost, a jedan od kompleksnijih primjera su samovozeći auti. Napretku su uvelike doprinijele i stohastičke optimizacijske metode koje su fokus ovog rada.

Istaknimo da je pripadnik šire obitelji metoda strojnog učenja i duboko učenje. Ono se temelji na umjetnim neuronskim mrežama i služi za tzv. reprezentativno učenje. Konkretno, ako su nam dani parovi ulaznih podataka x_i i opažanja y_i tražimo funkciju f_θ, iz parametrom θ definirane familije funkcija, koja na najbolji način „reprezentira" preslikavanje $x_i \mapsto y_i$. Jedan i drugi pojam ćemo precizno definirati u nastavku rada. U ovom trenutku se pouzdajemo u intuitivnu čitateljevu interpretaciju.

Istaknimo da tema ovog rada nije proučavanje pojedinih familija modela, već prezentacija metoda stohastičkog gradijentnog spusta na ilustrativnim primjerima koji u širem smislu spadaju u ovo područje. Konkretno, koncentrirat ćemo se na različite metode penalizacije u kontekstu linearne regresije te problemu upotpunjavanja matrica.

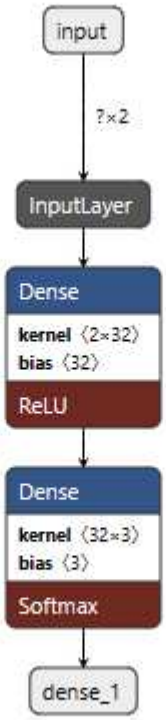
Zbog potpunosti i šireg konteksta rada, istaknimo da su se metode reprezentativnog učenja ukorijenile u mnoge sfere života i izvan matematike. Primjerice u proučavanje klimatskih promjena, prepoznavanje glasa, dizajn lijekova, analiza medicinskih dijagnoza te bi njegov daljnji razvoj dao veliki doprinos automatizaciji i općenito životnom standardu. Na slici 0.2 je dijagram koji prikazuje arhitekturu neuronske mreže. Neka je $\rho : \mathbb{R} \to \mathbb{R}$ neopadajuća funkcija i neka je $\boldsymbol{\rho} : \mathbb{R}^d \to \mathbb{R}^d$ funkcija definirana izrazom $\boldsymbol{\rho}(x) = \begin{bmatrix}\rho(x_1) & \cdots & \rho(x_d)\end{bmatrix}^T$. Za dane matrice $A_i \in \mathbb{R}^{d_i \times d_{i-1}}$ i vektore $b_i \in \mathbb{R}^{d_i}$, $i = 1, \cdots, r$ neuronska mreža je funkcija

$$f(\theta, x) = \left(\bigcirc_{i=1}^{r} \boldsymbol{\rho}(\boldsymbol{A_i} \cdot \boldsymbol{+b_i})\right)(x).$$

Ovdje $\bigcirc_{i=1}^{r}$ označava ulančanu kompoziciju funkcija, a niz θ, $\theta_i = (A_i, b_i)$, $i = 1, \cdots, r$ nazivamo arhitektura neuronske mreže. U slučaju kada se neuronske mreže koriste za rješavanje problema regresije, tražimo arhitekturu θ za koju će za zadani skup uređenih parova ulaza x_i i opažanja y_i greške aproksimacije modela $f(\theta, x_i) - y_i$, $i = 1, \cdot, n$ biti u nekom smislu minimalne. Za parametar θ ćemo općenito koristiti pojam numerička karakteristika modela.

Primarna metoda kojom ćemo tražiti numeričku karakteristiku θ za izabranu klasu modela, će biti metoda gradijentnog spusta s korekcijom adaptivnog momenta, popularno zvana Adam, koja „uči" kako odrediti veličinu koraka korekcije trenutne procijene parametra θ potrebnog za pouzdano pronalaženje lokalnog minimuma procijenjene greške aproksimacije modela.

U prvom poglavlju dajemo opis samih algoritama optimizacije baziranih na gradijentnom spustu, opisati njihovu funkcionalnost te ćemo ih prvo implementirati u programskom jeziku python, a potom ih analizirati na linearnoj regresiji . Tu je referentna literatura sljedeći članak [12] u kojem se može detaljnije pročitati o samim metodama. U drugom poglavlju implementiramo metodu Adam te proučavamo njenu konvergenciju.



Slika 0.2: Neuronska mreža; Izrađeno u Netronu [11]

Referenca je [8], dok će kod biti samostalno izrađen na osnovu pseudokoda. U trećem poglavlju, samu implementaciju algoritama koristimo na primjeru pogodno izabranih probleme linearne regresije kao metode strojnog učenja. U četvrtom poglavlju, istražujemo mogućnosti automatskog deriviranja koda biblioteki Autograd i Jax. U posljednjem, petom poglavlju, implementiramo rješenje problema upotpunjavanja (inpainting) slike pomoću matričnih faktorizacija koje otkrivaju rang matrice. Radi se o problemu rekonstruiranja (aproksimiranja) matrice, matricom niskog ranga. Dana matrica će u ovom slučaju predstavljati zapis slike, gdje oštećene dijelove podrazumijevamo kao nedostajući dio slike. Inpainting vršimo nenegativnom matričnom faktorizacijom matrice M na, obično, dvije matrice U i V, tako da su svi elementi tih triju matrica nenegativni. U ovom slučaju se za razliku od prethodnih regresijskih problema, radi o problemu optimizacije s ograničenjima. Matrica – tj slika – naime mora biti nenegativna po elementima. Mi se u ovom poglavlju

nećemo baviti problemom optimizacije s ograničenjem već ćemo bez dodatne analize konvergencije koristiti sljedeći plauzibilan pristup. Za danu nenegativnu matricu X_i računamo korigiranu matricu $\hat{X}_i$ nekom od gradijentnih metoda i onda sljedeću aproksimaciju X_{i+1} definiramo kao „najbližu" nenegativnu matricu matrici $\hat{X}_i$. Postupak kojim računamo takvu projekciju je često heuristički i nerijetko se za njega koristi pojam proročanstvo (oracle), vidi [6].

Poglavlje 1

Pregled optimizacijskih algoritama gradijentnog spusta

Opisat ćemo metode na kojima neće biti pretjeran naglasak ovog rada te ih implementirati, dok ćemo algoritme Adam i Adamax podrobnije proučiti u idućem poglavlju. Kasnije metode analiziramo na linearnoj regresiji.

1.1 Gradijentni spust

Uvodno

Potpoglavlje većinom bazirano na [14] i [3]. Neka je $f : \mathbb{R}^n \mapsto \mathbb{R}$ konveksna i diferencijabilna. Želimo riješiti problem $\min_{x\in\mathbb{R}^n} f(x)$, to jest, tražimo x^* takav da je $f(x^*) = \min f(x)$. Prvotno, uvodimo početni korak gradijentnog spusta za inicijalni $x_0 \in \mathbb{R}^n$ s veličinom koraka t:

$$x^{(k)} = x^{(k-1)} - t_k \nabla f(x^{(k-1)}), k = 1, 2, 3, ...$$

U svakoj iteraciji promatramo proširenje $f(y) \approx f(x) + \nabla f(x)^T(y - x) + \frac{1}{2t}\|y - x\|^2$ gdje smo s kvadratnom aproksimacijom zamijenili $\nabla^2 f(x)^T$ tako da je $f(x) + \nabla f(x)^T(y - x)$ linearna kombinacija za f, a $\frac{1}{2t}\|y - x\|^2$ izraz udaljenosti y od x s težinom $\frac{1}{2t}$. Sada biramo iduću točku $y = x^+$ kako bi minimizirali $x^+ = x - t\nabla f(x)$.

Definicija 1.1.1. *Uz prethodne pretpostavke za funkciju f, konveksnost i diferencijabilnost, dodatno pretpostavljamo* $\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|, \forall x, y \in \mathbb{R}^n$. *Kažemo da je* ∇f *Lipschitz-neprekidna s konstantom* $L > 0$.

Lipschitzova konstanta L daje mjeru „glatkoće" funkcije. Intuitivno, manja Lipschitz konstanta znači da se funkcija sporije mijenja i ima glađi oblik, dok veća Lipschitz konstanta znači da se funkcija brže mijenja i ima nazubljeniji oblik. Možemo ograničiti veličinu

gradijenta u bilo kojoj točki s L. Tada možemo odabrati dovoljno malu stopu učenja η tako da pravilo ažuriranja za spuštanje gradijenta jamči konvergenciju u lokalni minimum funkcije, čak i ako je funkcija nekonveksna.

Teorem 1.1.2. *Gradijentni spust s fiksnom veličinom koraka* $t \leq \frac{1}{L}$ *zadovoljava* $f(x^{(k)}) - f(x^*) \leq \frac{\|x^{(0)}-x^*\|^2}{2tk}$.

Primjer kriterija zaustavljanja su kada norma gradijenta funkcije cilja padne ispod određenog praga tolerancije što je obično indikator dobre aproksimacije lokalnog minimuma. Drugi primjer je kada promjena funkcije cilja između dvije uzastopne iteracije padne ispod određenog praga tolerancije što je indikator stabilnog rješenja. Na kriterij zaustavljanja može utjecati perturbacija funkcije cilja, koja se odnosi na uvođenje malih varijacija u funkciju cilja radi poboljšanja robusnosti i generalizacije optimizacijskog algoritma. Perturbacija funkcije cilja može spriječiti da algoritam zaglavi u lokalnim minimumima ili drugim suboptimalnim rješenjima uvođenjem slučajnosti u proces.

Neki pojmovi koji se pojavljuju u daljnjem tekstu

Prisjetimo se prvo gradijenta diferencijabilne funkcije. Navodimo propoziciju o postojanju parcijalne derivacije u točki, gdje je i definiran gradijent. Dokaz se može pronaći u [3].

Propozicija 1.1.3. *Neka je* $A \subseteq \mathbb{R}^n$ *i* $f : A \mapsto \mathbb{R}$ *te neka je* $v \in \mathbb{R}^n$ *jedinični vektor. Ako je funkcija diferencijabilna u točki c tada postoji* $\frac{\partial f}{\partial v}(c) = Df(c)v$.

Gradijent je vektor koji reprezentira diferencijal realne funkcije vektorske varijable i označavamo ga

$$\nabla f = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \dfrac{\partial f}{\partial x_2} \\ \vdots \\ \dfrac{\partial f}{\partial x_n} \end{bmatrix}$$

Napomena 1.1.4. *Gradijentom su određene najveće derivacije u smjeru u točki c, to jest, najbrži rast funkcije u smjeru vektora v.*

Gradijentni spust je iterativni optimizacijski algoritam prvog reda jer koristi informacije samo prve derivacije kako bi pripremio iduću iteraciju za pronalaženje lokalnog minimuma funkcije cilja $J(\theta)$ koja je parametrizirana parametrima modela $\theta \in \mathbb{R}^d$. Ideja je da u nadolazećim iteracijama mijenjamo parametre u smjeru suprotnom od smjera gradijenta jer

tako prema Napomeni 1.1.4 postižemo najveće spuštanje. Koristimo stopu učenja η kako bismo odredili veličinu koraka za pronalazak (lokalnog) minimuma. Metoda gradijentnog generira niz aproksimacija θ_i iterativno počevši od θ_0 po pravilu

$$\theta_{i+1} = \theta_i - \eta \nabla J(\theta_i).$$

Želimo minimizirati grešku između predviđene i stvarne vrijednosti. Pogreška je odstupanje odnosno razlika između te dvije vrijednosti.

Precizirajmo sada kako je definirana funkcija greške. Neka su dani parovi vrijednosti $x_i \mapsto \tilde{y}_i$ koji mogu predstavljati rezultate mjerenja. Za danu numeričku karakteristiku θ, koja opisuje familiju modela $y(\theta, \cdot)$, vrijedi $x_i \mapsto y_i(\theta) := y(\theta, x_i)$. **Funkcija gubitka** jest funkcija koja računa mjeru greške modela na testnom skupu $(x_i, \tilde{y}_i)$, $i = 1, \cdots, n$. U metodi iz trećeg poglavlja, linearnoj regresiji, će to biti srednja kvadratna pogreška (eng. mean squared error). Vrijednosti $\tilde{y}_i$ su definirane a priori i za neki θ vrijedi

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y_i(\theta) - \tilde{y}_i)^2. \tag{1.1}$$

```
def greška(x, y, theta):
    mse = np.sum((np.dot(x_i, theta)-y) ** 2) / (2 * broj)
```

Slika 1.1: Implementacija funkcija gubitka za linearnu regresiju u pythonu. U ovom slučaju je $y(\theta, x) = \theta^T x$, za $\theta \in \mathbb{R}^d$.

Vrste gradijentnog spusta

Razlikujemo tri vrste gradijentnog spusta prema količini podataka koje koristimo kako bismo izračunali gradijent ciljne funkcije $J(\theta)$. Ovisno o tome, balansiramo trošak između preciznosti promjene parametara i vremenu potrebnom da se promjena izvrši. Sada prikazujemo algoritme prilagođene aditivno separabilnoj strukturi funkcije gubitka.

Blokovni (batch) gradijentni spust

Radi se o standardnom algoritmu koji računa parametre modela θ za cijeli skup testnih primjera u jednom koraku. Funkcija gubitka se blokovski aproksimira te koordinate biramo tek u blokovima. Uzimamo prosjek gradijenata svih testnih primjera te zatim koristimo

aritmetičku sredinu gradijenata za ažuriranje naših parametara. Dakle, to je samo jedan korak gradijentnog spuštanja u jednoj epohi. Promjenu parametra dobijemo po zakonitosti

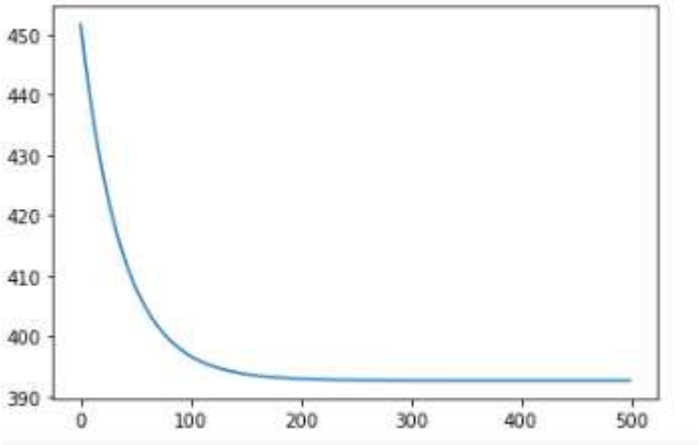
$$\theta = \theta - \eta \nabla_\theta J(\theta). \tag{1.2}$$

Mana mu je što je vrlo spor i ne dozvoljava ažuriranje modela „u hodu", to jest, ne mogu se naknadno proširiti testni primjeri modela ispitivanja. Što se konvergencije tiče, sigurno će konvergirati prema globalnom minimumu za konveksne funkcije greške te lokalnom minimumu za nekonveksne.

```
greške=[]
def blok(eta, x, y, koraci):
    broj = x.shape[0]
    theta = np.ones(2)
    x_trans = x.transpose()
    for i in range(0, koraci):
        gubitak = np.dot(x, theta) - y
        grad = np.dot(x_trans, gubitak)/ broj
        theta = theta - eta * grad
        greške.append(greška(x,y,theta))
    return theta
```

Slika 1.2: Implementacija blokovnog spusta

Koraci su broj epoha. Prije promjene parametara računa vektor gradijenta funkcije greške u unaprijed određenom broju koraka.



Slika 1.3: Greška u blokovnom spustu

Na x-osi se nalazi broj epoha, dok je na y-osi trošak. Vidimo da je greška velika u početku, ali kasnije se stabilizira.

Stohastički gradijentni spust

Promjenu parametra vrši za slučajno izabrani testni primjer x^i i y^i za razliku od blokovnog. U jednom prolazu razmatramo samo jedan – slučajno izabran – primjer te tražimo gradijent funkcije

$$J(\theta; x^i; y^i) = (y_i(\theta) - \tilde{y}_i)^2.$$

Trošak varira u odnosu na testne primjere te se neće nužno smanjiti. Dugoročno gledano se trošak smanjuje i korake metode definiramo izrazom

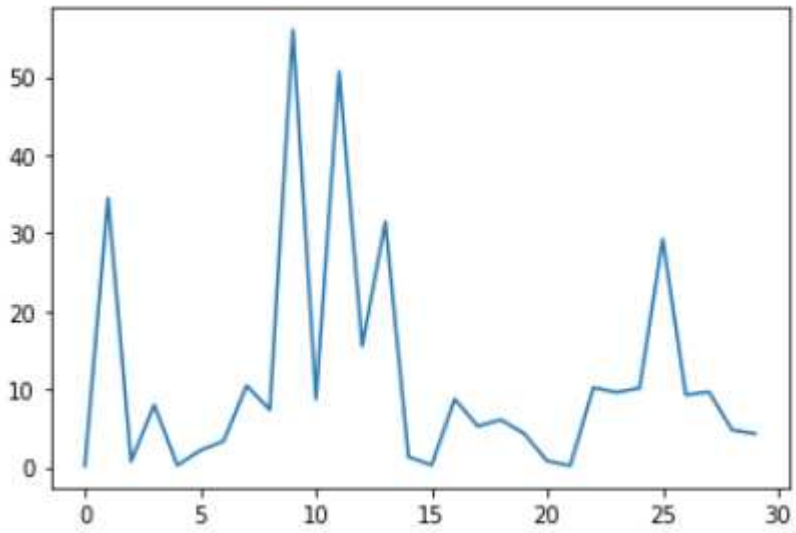
$$\theta = \theta - \eta \, \nabla_\theta J(\theta; x^i; y^i). \tag{1.3}$$

Izbjegava redundantne račune jer ne preračunava gradijente za slične testne primjere te omogućuje nadogradnju modela u vidu dodavanja novih testnih primjera naknadno. Postupnim smanjivanjem stope učenja η dolazi do sličnog konvergencijskog ponašanja kao blokovna. Zbog svoje flukturirajuće prirode, ima tendenciju skokova prema potencijalno boljem lokalnom minimumu u aktualnom koraku. Budući da je blokovski gradijent srednja vrijednosti gradijenata za pojedine primjere, onda je očekivanje slučajno izabranih $\nabla J(\theta; x^i; y^i)$ zapravo $\nabla J(\theta)$ i time ova stohastička aproksimacija zapravo aproksimira puni (blokovski) gradijent.

Zamijetimo miješanje (eng. shuffling) testnih primjera u svakoj iteraciji te na slici poziva greške, već u puno manjem broju iteracije svodi funkciju greške na minimum.

```python
greške=[]
def stohastični(eta, x, y, broj_epoha):
    broj = x.shape[0]
    theta = np.ones(2)
    velicina=len(y)
    for j in range(0, broj_epoha):
        for i in range(velicina):
            ind = np.random.randint(velicina)
            x_i = x[ind:ind+1]
            y_i = y[ind:ind+1]
        hipoteza = np.dot(x_i, theta)
        gubitak = hipoteza - y_i
        obj = np.sum(gubitak ** 2) / (2 * broj)
        greške.append(obj)
        grad = 2 * x_i.T.dot(gubitak)
        theta = theta - eta* grad
    return theta
```

Slika 1.4: Implementacija stohastičkog spusta



Slika 1.5: Greška u stohastičkom spustu

Mini-blokovni (batch) gradijentni spust

Ažurira parametre za svaki mini-blok od n testnih primjera koji je manji od stvarnog skupa podataka. U jednoj epohi za slučajno izabrani i računamo gradijent funkcije

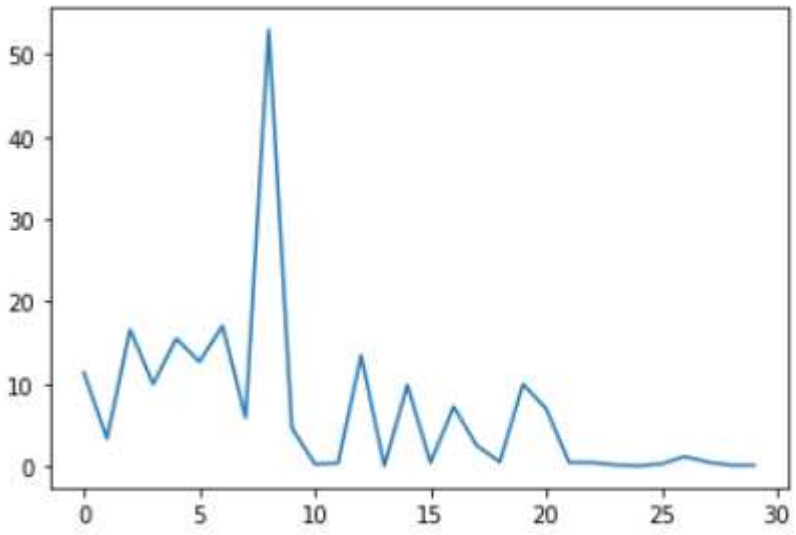
$$J(\theta; x^{(i:i+b)}; y^{(i:i+b)}) = \frac{1}{b}\sum_{k=i}^{b}(y_k(\theta) - \tilde{y}_k)^2$$

koja uz parametre θ, kao ulazni argument dobije i mini-blok širine $b \in \mathbb{N}$. $J(\theta; x^{(i:i+b)}; y^{(i:i+b)})$ je surogat funkcija, odnosno diferencijabilna funkcija koja aproksimira ciljnu funkciju koju želimo minimizirati i jednostavnija je za izračunati. U ovom slučaju služi kao zamjenska funkcija koja procjenjuje gradijent funkcije cilja na osnovu mini-bloka. Za mini-blokove veličine n računamo funkciju gubitka za svaki primjer u mini-bloku te zatim uzima srednju vrijednost tih gubitaka za procjenu funkcije cilja. Sada koristimo gradijent te zamjenske funkcije za ažuriranje parametara modela. Na taj način smanjuje varijancu kod parametarskih promjena što rezultira i stabilnijom konvergencijom te efikasno računa gradijent jer se mogu primijeniti visoko optimizirane matrične optimizacije dostupne u bibliotekama dubokog učenja. Ne garantira dobru konvergenciju. Kada govorimo o SGD (eng. stohastic gradient descent) metodama, podrazumijevamo ovu vrstu spusta te ćemo se u daljnjem tekstu i referirati na njega sa SGD. Razlika u kodu u odnosu na stohastički spust je u tome što se sada iteracije vrše samo na mini-blokovima veličine 50.

$$\theta = \theta - \eta\, \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}). \tag{1.4}$$

```
greške=[]
def mini_blok(eta, x, y, broj_epoha, velicina_bloka):
    broj = x.shape[0]
    theta = np.ones(2)
    velicina=len(y)
    for j in range(0, broj_epoha):
        ind = np.random.permutation(velicina)
        x_izmjena = x[ind]
        y_izmjena = y[ind]
        for i in range(0,velicina, velicina_bloka):
            x_i = x_izmjena[i:i+velicina_bloka]
            y_i = y_izmjena[i:i+velicina_bloka]
        hipoteza = np.dot(x_i, theta)
        gubitak = hipoteza - y_i
        obj = np.sum(gubitak ** 2) / (2 * broj)
        greške.append(obj)
        x_trans=x_i.transpose()
        grad = 2 * np.dot(x_trans,gubitak)
        theta = theta - eta* grad
    return theta
```

Slika 1.6: Implementacija mini-blokovnog spusta



Slika 1.7: Greška kod mini-blokovnog spusta

1.2 Optimizacijski algoritmi

Svrha ove podcjeline je ukratko opisati algoritme dubokog učenja koji će pokušati riješiti problem konvergencije stohastičkog gradijentnog spusta (SGD). Željene izmjene kod tog problema su sljedeće:

- Odabir adekvatnog faktora učenja η. Premalen η daje sporu konvergenciju, prevelik η može rezultirati divergencijom.
- Podešavanje vrijednosti praga na η može biti definirano samo prije početka algoritma.
- Oskudniji podaci bi trebali imati veću promjenu odjednom.
- Mogući problemi kod sedlastih točaka pa zapadne u neželjeni minimum.

U sljedećim fragmentima teksta v_t predstavlja vektor promjene, a već navedeni θ parametar nekog teorijskog modela dubokog učenja. Počnimo s prvim algoritmom koji pokušava ispraviti neke prethodno navedene mane SGD-a.

Moment

SGD oscilira na području ravninskih krivulja koje su strmije u prijelazu iz jedne dimenzije u drugu. Moment pomaže ubrzanju SGD-a u pravom smjeru i smanjuje oscilacije. Slikovito, moment je kao da pustimo loptu niz brdo nakon koje se ona kotrlja. Moment, u oznaci γ, je decimalna vrijednost manja od 1, obično postavljena na 0.9 koju dodajemo vektoru promjene prethodnog koraka u trenutni vektor promjene. Sam koncept na kojem se zasniva metoda ovisi o eksponencijalnom težinskom prosjeku pomoću kojeg smanjujemo oscilacije ciljne funkcije u potrazi za globalnim minimumom. Taj prosjek se dobije iz rekurzivne jednadžbe (1.5) pomoću momenta kojim postižemo veću „glatkoću" kod $J(\theta)$.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{1.5}$$

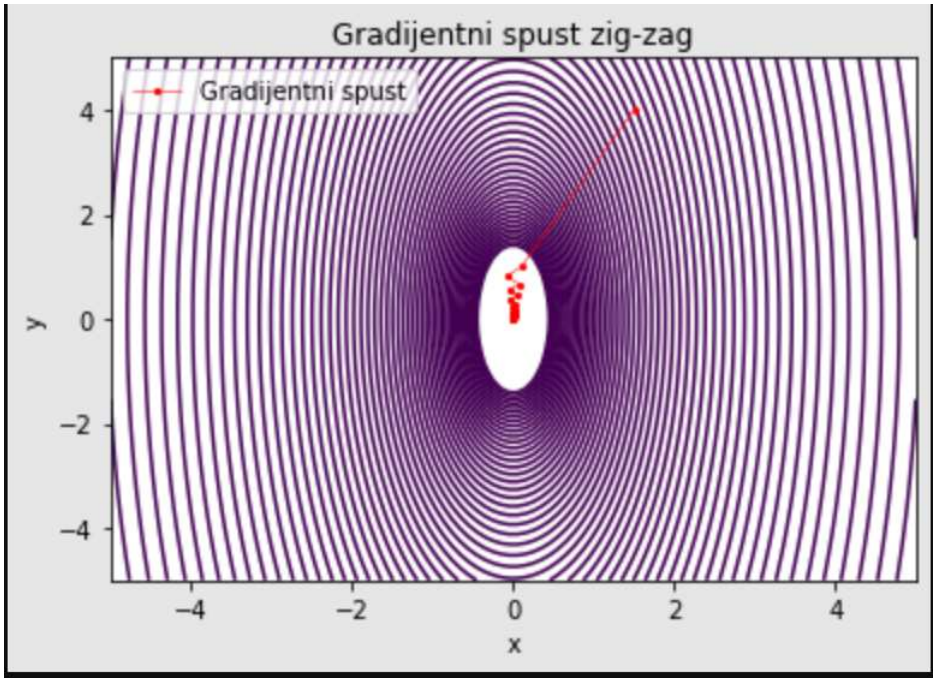
$$\theta = \theta - v_t. \tag{1.6}$$

Iznad su spomenuti vektor promjene te promjena parametra.

```
fja = lambda xy:(xy[0]**2)*(xy[1]**2)
theta = anp.array([0.1, 0.1])
pom=grad(fja)
pom(theta)
def sgd(f, theta, eta = 0.1, gama = 0.9, koraci = 200):
    v = np.zeros(len(theta))
    t = 0
    for t in range(koraci):
        pom = nd.Gradient(f)
        g = pom(theta)
        v= gama * v - (1.0 - gama)*g
        theta = theta + eta * v
    return theta
```

Slika 1.8: Kod spusta s momentom

Moment možemo shvatiti kao matematički alat koji koristi koncepte diferencijalnih jednadžbi i Markovljevih procesa za ubrzavanje optimizacije. Pravila promjene v_t i θ možemo zamisliti kao aproksimacije diskretnog vremena u diferencijalne jednadžbe kontinuiranog vremena koje opisuju gibanje čestice pod utjecajem sile te člana prigušenja. U smislu Markovljevih procesa, možemo moment promatrati kao jednostavan Markovljev lanac. U svakom vremenskom koraku, trenutno stanje, odnosno skup parametara θ_t se ažurira tranzicijskom funkcijom vektora brzine koje ovisi samo o trenutnom i prethodnom stanju v_{t-1}. Dakle, trenutno stanje ovisi samo o prethodnom stanju, dok su vjerojatnosti prijelaza određene vektorom brzine. Prikazat ćemo primjer spusta na zig-zag funkciji definiranoj formulom $f(x, y) = \frac{1}{2}(x^2 + b(y^2))$, gdje je $0 < b \leq 1$.



Slika 1.9: Primjer spusta s momentom na zig-zag funkciji za b=0.1

Nesterovljev akcelerirani gradijent (NAG)

Nesterovljev akcelerirani gradijent je metoda također bazirana na momentu korištenjem niza X_n, ali gleda unaprijed gdje bi se parametri modela θ mogli nalaziti pa računa gradijent. Drugim riječima prvo napravi skok u smjeru prethodno izračunatog gradijenta te potom računa gradijent gdje je završio pa ga ispravlja.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{1.7}$$

$$\theta_t = \theta_{t-1} - v_t. \tag{1.8}$$

Zeleni vektor je korekcija.

Slika 1.10: slikoviti opis NAG metode; Izvor: [12]

```
def NAG(f, theta, eta = 0.01, koraci = 50, mom=0.9):
    brzina = np.zeros(len(theta))
    grad = nd.Gradient(f)
    g = grad(theta)
    novi_par = theta - eta * g
    brzina = mom * brzina + (1 - mom) * (novi_par - theta)
    theta = theta - eta*g + mom*brzina
    return theta
```

Slika 1.11: Kod Nesterovljeve metode

Možemo ga promatrati kao vrstu prediktor-korektor metode. Dotične metode se koriste u brojnim područjima numeričke optimizacije te su učinkovite za probleme sa složenim objektnim funkcijama. Kao i ovdje, njihova je osnovna ideja prvo napraviti predviđanje rješenja temeljem trenutnog stanja i prethodnih derivacija te potom ispraviti predviđanje korištenjem stvarnih derivacija. Iterativno popravlja ova dva koraka do konvergencije k rješenju.

Adagrad

Idući korak je provesti veća ili manja ažuriranja, ovisna o značajnosti parametra, tj. frekvenciji njegova korištenja. Radi se o trećoj točkici prethodno spomenutih problema konvergencije SGD. Ovom problemu je doskočio Adagrad, optimizator koji za svaki parametar θ_i koristi drukčiju stopu učenja η. Neka je $g_{t,i}$ gradijent objektne funkcije J parametra θ_i na koraku t. Definiramo ga kao:

$$g_{t,i}(\theta) = \nabla_\theta J(\theta_{t,i}). \tag{1.9}$$

Sada je pravilo promjene parametra, ovisno i o modifikaciji stope učenja η, dano jednadžbom:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}. \tag{1.10}$$

Ovdje je $G_t \in R^{d*d}$ dijagonalna matrica kojoj je svaki element suma kvadrata gradijenata u ovisnosti o parametru θ_i. S Adagradom smo izbjegli ručno namještanje stope učenja η, koja je obično inicijalizirana na 0.01.

```
def adagrad(f, theta, eta = 0.01, epsilon = 1e-8, koraci = 100):
    G_ii=np.zeros((len(theta)))
    t = 0
    for t in range(koraci):
        grad = nd.Gradient(f)
        g = grad(theta)
        G_ii = G_ii + np.square(g)
        theta = theta - eta * g / (np.sqrt(G_ii) + epsilon)
    return theta
```

Slika 1.12: Kod Adagrad metode

Primjećujemo da je ovdje ključna promjena dodavanje matrice G_{ii} pomoću koje se vrši adaptacija stope učenja preko akumuliranih kvadrata gradijenata u svakoj iteraciji i u svakoj dimenziji. Pomoću ϵ izbjegavamo dijeljenje s 0, te je on obično reda 10^{-8}. Algoritam je najuspješniji na oskudnim podacima budući da za manje frekventne parametre sporije smanjuje stopu učenja, a za češće brže. Problem nastaje kada se stopa učenja naglo smanji te dođe do 0. Nakon toga model nije više u stanju učiti. Iduće 2 metode su se fokusirale na ispravak tog problema.

Adadelta

Metoda koja je nadogradnja algoritma Adagrad, cilj joj je smanjiti padajuću monotonost stope učenja η tako da ograniči prethodne gradijente s nekom fiksnom vrijednosti w. Suma gradijenata je rekurzivno definirana kao padajući (eng. decaying) prosjek svih prethodnih kvadratnih gradijenata. Propadajući prosjek je kontraintuitivni izraz (jer raste) za matematičko računanje prosjeka u kojemu se veća važnost pridodaje nedavno izračunatim vrijednostima ispitivanja, u ovom slučaju kvadrata gradijenata. Aktualni prosjek u koraku je dan formulom:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2. \tag{1.11}$$

γ je obično inicijalizirana na 0.9. Vektor promjene parametra modela i promjena parametra nakon iteracije algoritma su sljedeći:

$$\Delta\theta_t = -\eta g_{t,i} \tag{1.12}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t. \tag{1.13}$$

Umjesto akumulirane sume tijekom vremena, računamo akumulirani eksponencijalni propadajući prosjek kvadratnih gradijenata.

```
def adadelta(f, theta, gama = 0.9, epsilon = 1e-8):
    delta=np.zeros((len(theta)))
    prosjek_g=np.zeros((len(theta)))
    prosjek_par=np.zeros((len(theta)))
    grad = nd.Gradient(f)
    g = grad(theta)
    prosjek_par = gama*prosjek_par + (1- gama)*g**2
    delta = -np.sqrt(prosjek_g + epsilon) / np.sqrt(prosjek_par + epsilon) * g
    prosjek_g = gama*prosjek_g + (1-gama)*(delta**2)
    theta = theta + delta
    return theta
```

Slika 1.13: Kod Adadelta

RMSProp

Metoda koja je nastajala u slično vrijeme kad i Adadelta. RMSProp (skraćeno od eng. root mean squared propagation) je identičan prvom vektoru promjene Adadelte, vidljivo u jednadžbama:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{1.14}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t. \tag{1.15}$$

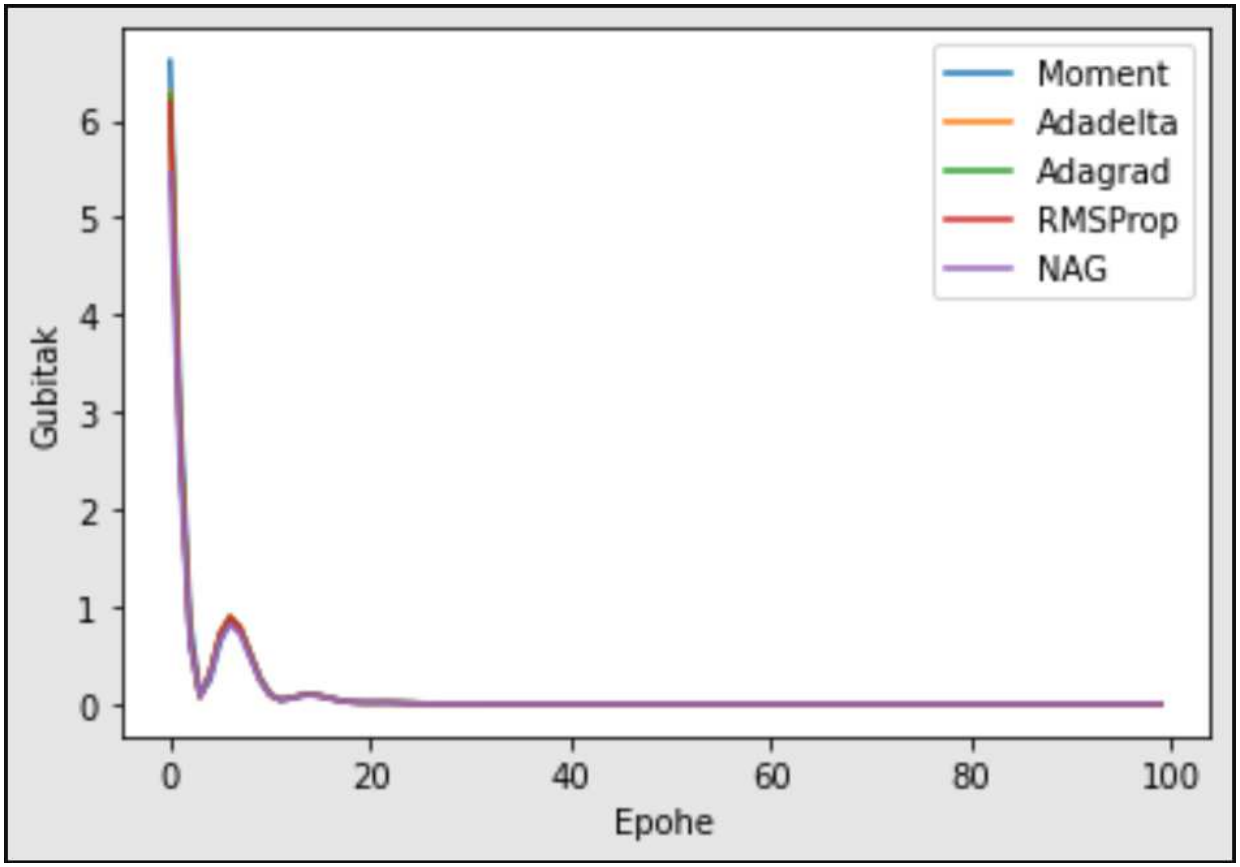
RMSProp koristi drugi moment (necentirana varijanca), odnosno eksponencijalni opadajući prosjek prošlih gradijenata kako bi imao ubrzanje u odnosu na Adagrad. Adam je idući korak evolucije optimiziranja, koristeći također i prvi moment. RMSProp dijeli stopu učenja s pomičnim prosjekom kvadrata gradijenata, umjesto da ga dodaje. To pomaže smanjiti utjecaj akumuliranih gradijenata tijekom vremena i poboljšati konvergenciju optimizatora.

```
def rmsprop(f, theta, koraci=100, eta=0.1, gamma=0.9, epsilon=10**-8 ):
    v_h = np.zeros(len(theta))
    for t in range(koraci):
        grad = nd.Gradient(f)
        g = grad(theta)
        v_h = gamma * v_h + (1-gamma)*(np.square(g))
        theta = theta - eta * g / (np.sqrt(v_h) + epsilon)
    return theta
```

Slika 1.14: Kod RMSProp-a

Međusobni odnosi algoritama

Ova podcjelina prikazuje odnos spomenutih algoritama prema random testnim podacima. Rezultati su dobiveni metodom linearne regresije. Funkcija greške je ponovno MSE. Nesterovljev algoritam se prilično poklapa s momentom pa nije najvidljiviji. Na ostalim metodama primjećujemo veće pravilnosti sa standardnim oscilacijama, a prednjači RMSProp metoda s najmanjim gubicima. Donja slika je izrađena pomoću Pytorch biblioteke o kojoj će kasnije biti više rečeno, a navedeni optimizatori su kasnije uspoređeni bez pomoći iste.



Slika 1.15: Usporedba funkcije greške među algoritmima

Prednosti i mane pojedinih metoda

- Moment- brzo se kreće kroz „plitka" područja funkcije cilja i radi veće skokove prema minimumu. Dobro funkcionira u problemima konveksne i glatke optimizacije. Može imati poteškoće u navigaciji oštrih zavoja,

- Adadelta- konvergira brže od mnogih drugih algoritama, radi dobro za velike skupove podataka,
- Adagrad- radi dobro s rijetkim podacima te može brzo konvergirati u lokalni minimum. Može biti osjetljiv na početnu stopu učenja η te kada je ista premala ima problema s konvergencijom,
- RMSProp- dobro se nosi s oscilirajućim gradijentima i brzo konvergira. Ima problema s nekonveksnim funkcijama cilja,
- NAG- slično kao moment, ali brže konvergencije.

Poglavlje 2

Adaptivna procjena momenta (Adam)

Adam je učinkovit i primjenjiv na širokom rasponu problema dubokog učenja koji koriste veliku količinu podataka ili parametara. Sposoban je raditi s rijetkim gradijentima, kompjuterski je efikasna metoda, koristi malo memorije te su mu predloženi hiperparametri dobri te neovisni o naknadnom podešavanju. Brzina učenja se automatski prilagođava za svaki parametar što dovodi do brže konvergencije i boljih performansi. Efikasna je metoda jer mu je dovoljna samo informacija gradijenta prvog reda.

2.1 Preliminarnosti

Definicije su preuzete iz [1] i [13].

Definicija 2.1.1. *Neka je X diskretna slučajna varijabla s funkcijom gustoće f. Ako vrijedi* $\sum_{x\in R}|x|f(x) < \infty$*, onda kažemo da X ima matematičko očekivanje koje definiramo kao*

$$E[X] = \sum_{x\in R} xf(x).$$

Definicija 2.1.2. *Neka je X diskretna slučajna varijabla s funkcijom gustoće f i matematičkim očekivanjem E[X]. Varijanca od X se definira kao* $VarX := E((X - E[X])^2)$*.*

Definicija 2.1.3. *Neka je V vektorski prostor nad* $\mathbb{R}$*. Skalarni produkt nad V je preslikavanje* $\langle\cdot\ ,\ \cdot\rangle : VxV \mapsto \mathbb{R}$ *sa sljedećim svojstvima:*

- $\langle x\ ,\ x\rangle \geq 0,\ \forall x \in V,$
- $\langle x\ ,\ x\rangle = 0 \iff x = 0,$
- $\langle x_1 + x_2\ ,\ y\rangle = \langle x_1\ ,\ y\rangle + \langle x_2\ ,\ y\rangle,\ \forall x_1, x_2, y \in V,$

- $\langle \alpha x\ ,\ y\rangle = \alpha\langle x\ ,\ y\rangle,\ \forall \alpha \in \mathbb{R}\ \forall x, y \in V,$
- $\langle x\ ,\ y\rangle = \overline{\langle y\ ,\ x\rangle},\ \forall x, y \in V.$

Definicija 2.1.4. *Vektorski prostor nad kojim je definiran skalarni produkt zovemo unitaran vektorski prostor. Neka je V unitaran prostor. Norma na V je funkcija* $\|\cdot\| : V \to \mathbb{R}$ *definirana s* $\|x\| = \sqrt{\langle x\ ,\ x\rangle}$.

Uvodno o metodi

Velika većina ovog poglavlja se referira na članak [8]. Adam je metoda koja je bazirana na adaptivnim izračunima momenata nižih redova. Računa stopu učenja η za svaki parametar modela ispitivanja. Nadogradnja je Adadelte i RMSProp-a u smislu da pamti eksponencijalne opadajuće prosjeke prošlih gradijenata m_t, slično momentu:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{2.1}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2. \tag{2.2}$$

m_t je prvi moment (srednja vrijednost), a v_t je drugi moment (necentrirana varijanca) gradijenata. Potrebne su preinake za ta dva vektora jer su devijantni prema 0, stoga koristimo

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.3}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{2.4}$$

Sama promjena vektora parametara dana je s:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t. \tag{2.5}$$

Inicijalne vrijednosti su $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, gdje su β_1 i β_2 eksponencijalne stope opadanja kod računanja momenata.

Korekcija inicijalnog odstupanja

U slučaju rijetkih gradijenata, kako bismo došli do pouzdane procjene momenta, potrebno je pronaći prosjek preko brojnih gradijenata postavljanjem male vrijednosti β_2. U ovom slučaju bi manjak naslovne korekcije doveo do velikih početnih koraka što želimo izbjeći. Slijedi izvod za procjenu drugog momenta.
Neka je g gradijent objektne stohastičke funkcije f. Želimo procijeniti drugi moment pomoću eksponencijalnog prosjeka kvadratnog gradijenta sa stopom propadanja β_2. Neka

su $g_1, ..., g_T$ gradijenti pripadnog vremenskog odmaka. Eksponencijalni prosjek inicijaliziramo kao $v_0 = 0$. Promjenu prosjeka na vremenskom odmaku t možemo zapisati kao:

$$v_t = (1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-1} g_i^2.$$

Cilj je saznati kako je $E[v_t]$, što je očekivana vrijednost prosjeka u koraku t, povezana s pravim drugim momentom $E[g_t^2]$ kako bismo ispravili odstupanje.

$$\begin{aligned} E[v_t] &= E((1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-1} g_i^2) \\ &= E[g_t^2](1 - \beta_2) \sum_{i=1}^{t} \beta_2^{t-1} + \zeta \\ &= E[g_t^2](1 - \beta_2^t) + \zeta \end{aligned}$$

Ukoliko je drugi moment $E[g_t^2]$ stacionaran, vrijedi $\zeta = 0$, inače je ζ postavljena na male vrijednosti kako bi smo odabrali β_1 tako da prosjek pridaje male težine davnim gradijentima. Preostaje izraz $(1 - \beta_2^t)$ koji postoji zbog inicijalizacije $v_0 = 0$. Zato ga koristimo kako bismo ispravili odstupanje. Analogno se izvodi prvi moment.

2.2 Konvergencija

Neka je $f_1(\theta), f_2(\theta), ..., f_T(\theta)$ niz proizvoljnih i nepoznatih konveksnih funkcija gubitaka. U svakom trenutku t, cilj je predvidjeti parametar θ_t i procijeniti ga na prethodnim nepoznatim funkcijama gubitka f_t. Budući da je priroda niza nepoznata unaprijed, procjenjujemo algoritam koristeći sumu svih prethodnih razlika između aktualnog predviđanja $f_t(\theta_t)$ i najboljeg parametra fiksne točke θ^* iz promatranog skupa parametara X za sve prethodne korake. Tu sumu nazivamo žaljenje (eng. regret) te ju definiramo:

$$R(T) = \sum_{t=1}^{T} \left[(f_t(\theta_t) - f_t(\theta^*) \right].$$

Definiramo $\theta^* = arg\,\min_{\theta \in X} \sum_{t=1}^{T} f_t(\theta)$. Gornja ograda je $O(\sqrt{T})$ za gore definiranu funkciju, što je usporedivo s najboljim ogradama za općenite probleme konveksnog učenja.

Dokaz konvergentnosti

Definicija 2.2.1. *Funkcija $f : \mathbb{R}^d \mapsto \mathbb{R}$ je konveksna ako za svaki $x, y \in \mathbb{R}^d$, te za svaki $\lambda \in [0, 1]$ vrijedi*

$$\lambda f(x) + (1 - \lambda) f(y) \geq f(\lambda x + (1 - \lambda) y). \tag{2.6}$$

Konveksna funkcija može biti ograničena odozdo s hiperravninom na svojoj tangenti. Za dokaz konvergencije bit će nam potrebne sljedeće 3 tehničke leme.

Lema 2.2.2. *Ako je funkcija $f : \mathbb{R}^d \mapsto \mathbb{R}$ konveksna, tada za svaki $x, y \in \mathbb{R}^d$ vrijedi*

$$f(y) \geq f(x) + \nabla f(x)^T (y - x). \tag{2.7}$$

Ova lema služi za gornju omeđenost te će u glavnom dokazu hiperavnina biti zamijenjena s Adamovim pravilima promjene.

Lema 2.2.3. *Neka je $g_t = \nabla f_t(\theta_t)$ i $g_{1:t}$ vektor koji sadrži gradijente svih iteracija do t od i-te dimenzije te je ograničen, $\|g_t\|_2 \leq G$, $\|g_t\|_\infty \leq G_\infty$. Tada, $\sum_{t=1}^{T} \sqrt{\frac{g_{t,i}^2}{t}} \leq 2G_\infty \|g_{1:T,i}\|_2$*

Lema 2.2.4. *Neka $\gamma \triangleq \frac{\beta_1^2}{\sqrt{\beta_2}}$. Za $\beta_1, \beta_2 \in [0, 1>$ takvi da zadovoljavaju $\frac{\beta_1^2}{\sqrt{\beta_2}} < 1$ i ograničeni g_t, $\|g_t\|_2 \leq G$, $\|g_t\|_\infty \leq G_\infty$ vrijedi sljedeća nejednakost*

$$\sum_{t=1}^{T} \sqrt{\frac{\hat{m}_{t,i}^2}{\sqrt{t\hat{v}_{t,i}}}} \leq \frac{2}{1-\gamma} \frac{1}{\sqrt{1-\beta_2}} \|g_{1:T,i}\|_2. \tag{2.8}$$

Teorem 2.2.5. *Pretpostavimo da funkcija f_t ima ograničene gradijente, $\|\nabla f_t(\theta)\|_2 \leq G$, $\|\nabla f_t(\theta)\|_\infty \leq G_\infty$ za svaki $\theta \in \mathbb{R}^d$ te svaka udaljenost između bilo kojeg parametra θ_t kojeg će generirati Adam biti ograničena, $\|\theta_n - \theta_m\|_2 \leq D$, $\|\theta_m - \theta_n\|_\infty \leq D_\infty$ za svaki $m, n \in 1, ..., T$ i $\beta_1, \beta_2 \in [0, 1>$ zadovoljava $\frac{\beta_1^2}{\sqrt{\beta_2}} < 1$. Neka $\alpha_t = \frac{\alpha}{\sqrt{t}}$ i $\beta_{1,t} = \beta_1 \lambda^{t-1}$, $\lambda \in (0, 1)$. Adam postiže za svaki $T \geq 1$ sljedeće:*

$$R(T) \leq \frac{D^2}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \sqrt{T\hat{v}_{T,i}} + \frac{\alpha(\beta_1+1)G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2} \sum_{i=1}^{d} \|g_{1:T,i}\|_2 + \sum_{i=1}^{d} \frac{D_\infty^2 G_\infty \sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\lambda)^2}.$$

Dokaz. Koristeći Lemu 2.1.2 imamo:

$$f_t(\theta_t) - f_t(\theta^*) \leq g_t^T(\theta_t - \theta^*) = \sum_{i=1}^{d} g_{t,i}(\theta_{t,i} - \theta_{,i}^*).$$

Iz pravila promjene parametara imamo,

$$\begin{aligned} \theta_{t+1} &= \theta_t - \alpha_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t}} \\ &= \theta_t - \frac{\alpha_t}{1-\beta_1^t} \left(\frac{\beta_{1,t}}{\sqrt{\hat{v}_t}} m_{t-1} + \frac{1-\beta_{1,t}}{\sqrt{\hat{v}_t}} g_t \right). \end{aligned}$$

Primarno nam je imati naglasak na i-tu dimenziju u vektoru parametra $\theta_t \in R^d$. Oduzmimo skalar $\theta_{,i}$ te kvadrirajmo obje strane gornje jednakosti.

$$(\theta_{t+1} - \theta_{,i})^2 = (\theta_t - \theta^*_{,i})^2 - \frac{2\alpha_t}{1-\beta_1^t}(\frac{\beta_1^t}{\sqrt{\hat{v}_{t,i}}} m_{t-1,i}) + (1 - \frac{\beta_{1,t}}{\sqrt{\hat{v}_{t,i}}} g_{t,i})(\theta_t - \theta^*_{,i})) + \alpha_t^2 (\frac{\hat{m}_{t,i}}{\sqrt{\hat{v}_{t,i}}})^2.$$

Koristeći Youngovu nejednakost možemo preurediti gornju jednakost te pokazati

$\sqrt{\hat{v}_{t,i}} = \frac{\sqrt{\sum_{j=1}^t (1-\beta_2)\beta_2^{t-j} g_{j,i}^2}}{\sqrt{1-\beta_2^t}} \leq \|g_{1:t,i}\|_2$ i $\beta_{1,t} \leq \beta_1$. Tada slijedi

$$\begin{aligned} g_{t,i}(\theta_{t,i} - \theta^*_{,i}) &= \frac{(1-\beta_1^t)\sqrt{\hat{v}_{t,i}}}{2\alpha_t(1-\beta_{1,t})}((\theta_{t,i} - \theta^*_{,i})^2 - (\theta_{t+1,i} - \theta^*_{,i})^2) + \frac{\beta_{1,t}}{(1-\beta_{1,t})} \frac{\hat{v}_{t-1,i}^{1/4}}{\sqrt{\alpha_{t-1}}} (\theta^*_{,i} - \theta_{t,i}) \sqrt{\alpha_{t-1}} \frac{m_{t-1,i}}{\hat{v}_{t-1,i}^{1/4}} \\ &+ \frac{\alpha_t(1-\beta_1^t)\sqrt{\hat{v}_{t,i}}}{2(1-\beta_{1,t})} (\frac{\hat{m}_{t,i}}{\sqrt{\hat{v}_{t,i}}})^2 \\ &\leq \frac{1}{2\alpha_t(1-\beta_1)}((\theta_{t,i} - \theta^*_{,i})^2 - (\theta_{t+1,i} - \theta^*_{,i})^2)\sqrt{\hat{v}_{t,i}} + \frac{\beta_{1,t}}{2\alpha_{t-1}(1-\beta_{1,t})}(\theta^*_{,i} - \theta_{t,i})^2 \sqrt{\hat{v}_{t-1,i}} \\ &+ \frac{\beta_1 \alpha t - 1}{2(1-\beta_1)} \frac{m_{t-1,i}^2}{\sqrt{\hat{v}_{t-1,i}}} + \frac{\alpha_t}{2(1-\beta_1)} \frac{m_{t,i}^2}{\sqrt{\hat{v}_{t,i}}}. \end{aligned}$$

Sada primjenjujemo Lemu 2.1. na gornju nejednakost i računamo donju ogradu sumirajući sve dimenzije $i \in 1, ..., d$ na gornjoj ogradi $f_t(\theta_t) - f_t(\theta^*)$ na niz konveksnih funkcija za $t \in 1, ..., T$:

$$\begin{aligned} R(T) &\leq \sum_{i=1}^d \frac{1}{2\alpha_1(1-\beta_1)}(\theta_{1,i} - \theta^*_{,i})^2 \sqrt{\hat{v}_{1,i}} + \sum_{i=1}^d \sum_{t=2}^T \frac{1}{2(1-\beta_1)}(\theta_{t,i} - \theta^*_{,i})^2 (\frac{\sqrt{\hat{v}_{t,i}}}{\alpha_t} - \frac{\sqrt{\hat{v}_{t-1,i}}}{\alpha_{t-1}}) \\ &+ \frac{\beta_1 \alpha G_\infty}{(1-\beta_1)(\sqrt{1-\beta_2})(1-\gamma)^2} \sum_{i=1}^d \|g_{1:T,i}\|_2 + \frac{\alpha G_\infty}{(1-\beta_1)(\sqrt{1-\beta_2})(1-\gamma)^2} \sum_{i=1}^d \|g_{1:T,i}\|_2 \\ &+ \sum_{i=1}^d \sum_{t=2}^T \frac{\beta_{1,t}}{2\alpha_t(1-\beta_{1,t})}(\theta^*_{,i} - \theta_{t,i})^2 \sqrt{\hat{v}_{t,i}}. \end{aligned}$$

Iz pretpostavke $\|\theta_n - \theta_m\|_2 \leq D$, $\|\theta_m - \theta_n\|_\infty \leq D_\infty$ slijedi:

$$
\begin{aligned}
R(T) &\leq \frac{D^2}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \sqrt{T\hat{v}_{T,i}} + \frac{(1+\beta_1)\alpha G_\infty}{(1-\beta_1)(\sqrt{1-\beta_2})(1-\gamma)^2} \sum_{i=1}^{d} \|g_{1:T,i}\|_2 \\
&+ \frac{D_\infty^2}{2\alpha} \sum_{i=1}^{d} \sum_{t=1}^{t} \frac{\beta_{1,t}}{(1-\beta_{1,t})} \sqrt{t\hat{v}_{t,i}} \\
&\leq \frac{D^2}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \sqrt{T\hat{v}_{T,i}} + \frac{(1+\beta_1)\alpha G_\infty}{(1-\beta_1)(\sqrt{1-\beta_2})(1-\gamma)^2} \sum_{i=1}^{d} \|g_{1:T,i}\|_2 \\
&+ \frac{D_\infty^2 G_\infty^2 \sqrt{1-\beta_2}}{2\alpha} \sum_{i=1}^{d} \sum_{t=1}^{t} \frac{\beta_{1,t}}{(1-\beta_{1,t})} \sqrt{t}.
\end{aligned}
$$

Za zadnji sumand koristimo gornju ogradu aritmetičko-geometrijskog reda:

$$
\begin{aligned}
\sum_{t=1}^{t} \frac{\beta_{1,t}}{(1-\beta_{1,t})} \sqrt{t} &\leq \sum_{t=1}^{t} \frac{1}{(1-\beta_1)} \lambda^{t-1} \sqrt{t} \\
&\leq \sum_{t=1}^{t} \frac{1}{(1-\beta_1)} \lambda^{t-1} t \\
&\leq \frac{1}{(1-\beta_1)(1-\lambda^2)}.
\end{aligned}
$$

Iz svega ovoga slijedi donja ograda metode dana u tvrdnji teorema. □

2.3 Implementacija

Pseudokod po kojem ćemo implementirati Adama u Pythonu:

Algoritam 1 Adam

Ulaz: α ▷ veličina koraka
Ulaz: β_1, β_2 ▷ hiperparametri za procjenu momenta
Ulaz: $f(\theta)$ ▷ funkcija cilja s parametrima θ
Ulaz: θ_0 ▷ inicijalni vektor parametara
Izlaz: θ_t ▷ rezultirajući parametri
$m_0 \leftarrow 0$ ▷ inicijalizacija vektora prvog momenta
$v_0 \leftarrow 0$ ▷ inicijalizacija vektora drugog momenta
$t \leftarrow 0$ ▷ inicijalizacija vremenskog pomaka
dok θ_t ne konvergira **radi**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ ▷ izračunaj gradijent ciljne funkcije u koraku t
$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ izračunaj nekorigiranu procjenu prvog momenta
$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ ▷ izračunaj nekorigiranu procjenu drugog momenta
$\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ ▷ izračunaj korigiranu procjenu prvog momenta
$\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ ▷ izračunaj korigiranu procjenu drugog momenta
$\hat{\theta}_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ ▷ ažuriraj parametre
kraj dok

```
def adam(f, theta, eta = 1e-3, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8, koraci = 100):
    m = np.zeros(len(theta))
    v = np.zeros(len(theta))
    t = 0
    for t in range(koraci):
        grad = nd.Gradient(f)
        g = grad(theta)
        m = beta1 * m + (1-beta1)*g
        v = beta2 * v + (1-beta2)*(np.square(g))
        m_h = m/(1-beta1**(t+1))
        v_h = v/(1-beta2**(t+1))
        theta = theta - eta * m_h / (np.sqrt(v_h) + epsilon)
    return theta
```

Slika 2.1: Adam

2.4 Proširenje Adamax

Adamax je varijanta Adama. Član v_t u Adamovom pravilu, skalira gradijent inverzno propocionalno L^2 normi prošlih gradijenata i trenutnog gradijenta. Generalizacijom L^p prostora, dobijemo nepouzdane varijante Adama, ali posebni slučaj kad $p \to \infty$ je stabilan te se radi o prethodno navedenom proširenju Adamax. Slijedi izvod algoritma:

$$v_t = \beta_2^p V_{t-1} + (1-\beta_2^p)|g_t|^p = (1-\beta_2^p)\sum_{i=1}^{t}\beta_2^{p(t-1)}|g_i|^p.$$

Definirajmo:

$$u_t = \lim_{p\to\infty}(v_t)^{\frac{1}{p}}.$$

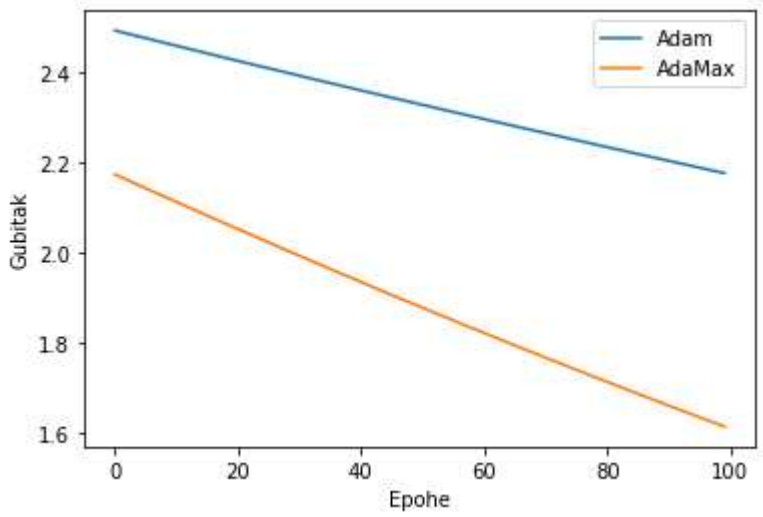
Sada vrijedi:

$$\begin{aligned}
u_t &= \lim_{p\to\infty}(v_t)^{\frac{1}{p}} \\
&= \lim_{p\to\infty}\left((1-\beta_2^p)\sum_{i=1}^{t}\beta_2^{p(t-1)}|g_i|^p\right)^{\frac{1}{p}} \\
&= \lim_{p\to\infty}(1-\beta_2^p)^{\frac{1}{p}}\left(\sum_{i=1}^{t}\beta_2^{p(t-1)}|g_i|^p)\right)^{\frac{1}{p}} \\
&= \lim_{p\to\infty}\left(\sum_{i=1}^{t}\beta_2^{(t-1)}|g_i|^p)\right)^{\frac{1}{p}} \\
&= \max(\beta_2^{(t-1)}|g_1|, \beta_2^{(t-2)}|g_2|, ..., \beta_2|g_{t-1}|, |g_t|).
\end{aligned}$$

Iz ovoga proizlazi rekurzivna formula $u_t = \max(\beta_2 u_{t-1}, |g_t|)$ s početnom vrijednosti $u_t = 0$. Oba algoritma su prilično jednostavna za implementirati u Pythonu, razlikuju se u tome što zbog max funkcije prilikom računanja u_t izbjegavamo devijantnost prema 0 kao kod vektora v_t i m_t u Adamu, stoga nije potrebna korekcija funkcije greške na svakom trening skupu. Inicijalne vrijednosti hiperparametara su identične Adamu.

```
def adamax(f, theta, koraci=100, eta=2e-3, beta1=0.9, beta2=0.999, epsilon=1e-8):
    mom = np.zeros(2)
    nor = np.zeros(2)
    t = 0
    for t in range(koraci):
        grad = nd.Gradient(f)
        g = grad(theta)
        mom = beta1 * mom + (1 - beta1) * g
        nor = beta2 * nor + (1 - beta2) * abs(g)
        theta = theta - eta * mom / (nor + epsilon)
    return theta
```

Slika 2.2: Adamax



Slika 2.3: Usporedba Adam i Adamax metode

Usporedba

Općenito su obje metode efikasne za linearnu regresiju, međutim, u ovom slučaju Adamax je brži na random generiranom skupu podataka. Hiperparametri su identični navedenima u članku. Što se tiče memorije za pohranu, Adamaxu je potrebno nešto više jer sprema dodatne prosjeke aposlutnih vrijednosti gradijenata, no količina memorije koju u tom slučaju upotrebljava je zanemariva. Izbor ovisi najviše o skupu podataka promatranog problema.

2.5 Nadam

Prethodno je rečeno da je Nesterovljev akcelerirani gradijent bolji od običnog momenta. Kombiniranjem Adam metode i NAG-a dobijemo naslovni optimizator punog imena Nesterovljev akcelerirani adaptivni procjenjitelj momenta. Kako bismo dodali NAG u Adama, ponovno je potrebno izmijeniti neka pravila, odnosno, u ovom slučaju izvršiti ekspanziju. Izmjena parametara ispitivanja poprima jednakost:

$$\theta_{t+1} = \theta_t - (\gamma m_{t-1} + \eta g_t).$$

Sada moment u ovom algoritmu zahtijeva korak u smjeru prethodnog momenta i smjera trenutnog gradijenta. Kako bismo došli do potpunog NAG-a unutar Adama potrebna je izmjena i gradijenta. Prema prijedlogu matematičara Dozata, umjesto da dva puta radimo korak momenta, (jednom za gradijent g_t i drugi put za težine θ_{t+1}, možemo izravno procijeniti vektor momenta kako bismo ažurirali trenutne težine:

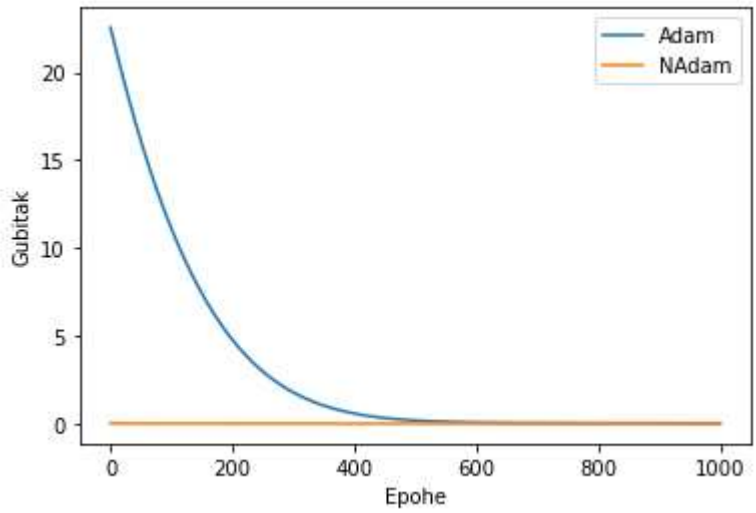
$$\theta_{t+1} = \theta_t - (\gamma m_t + \eta g_t).$$

Iz prethodne jednadžbe je vidljiva promjena iz m_{t-1} u m_t. Sada samo preostaje u ažuriranju Adama izmijeniti njegov moment m i $\hat{m}$ tako da umjesto prethodnog koristi trenutni vektor momenta. Iz svega ovog slijede pravila promjene za Nadam:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}).$$

```
def nadam(f, theta, koraci=100, eta=1e-3, beta1=0.9, beta2=0.999, epsilon=1e-8):
    m = np.zeros(len(theta))
    v = np.zeros(len(theta))
    t = 0
    for t in range(koraci):
        grad = nd.Gradient(f)
        g = grad(theta)
        m = beta1 * m + (1 - beta1) * g
        v = beta2 * v + (1 - beta2) * (g ** 2)
        m_h = m / (1 - beta1 ** (t+1))
        v_h = v / (1 - beta2 ** (t+1))
        theta = theta - eta * (beta1 * m_h + (1 - beta1) * g / (1 - beta1 ** (t+1))) / (np.sqrt(v_h) + epsilon)
    return theta
```

Slika 2.4: Nadam



Slika 2.5: Usporedba Adam i Nadam metode

Usporedba

Zbog inkorporiranog NAG-a, u ovom primjeru primjećujemo bržu konvergenciju i stabilnost kod Nadam metode. Glede korekcije inicijalnog odstupanja, Nadam radi isto što i Adam, ali također vrši i korekciju pristranosti na pomični prosjek kvadratnih gradijenata što smanjuje utjecaj početnih epoha na promjenu parametara. U ovom primjeru to ne primjećujemo, no Nadam je uspješniji i u sedlastim točkama prilagođavanjem stope učenja i momenta.

Poglavlje 3

Primjena optimizatora na metodu strojnog učenja

3.1 Metoda linearne regresije

Formalna definicija

Definicija preuzeta iz [17]. S obzirom na skup podataka $\{y_i, x_{i,1}, ..., x_{i,p}\}_{i=1}^{n}$ od n statističkih jedinica, linearni regresijski model pretpostavlja da je odnos između zavisne varijable y i vektora regresora x linearan. Odnos je modeliran kroz izraz poremećaja ili varijablu pogreške ϵ — neopaženu slučajnu varijablu koja dodaje „šum" linearnom odnosu između zavisne varijable i regresora. Tako model poprima oblik

$$y_i = \beta_0 + \beta_1 x_{i,1} + ... + \beta_p x_{i,p} + \epsilon_i, i = 1, 2, ..., n.$$

Uvodno

U suštini se radi o statističkoj metodi koja je primjenjiva i u strojnom učenju. Prognozer varijabla je neovisna varijabla koja se ne mijenja prilikom promjene ostalih, ispitnih, varijabli. Na osnovu fluktuacija neovisne varijable se mijenjaju ovisne varijable. Sami linearni regresijski model, temeljem odgovora ili ishoda analizirane varijable predviđa vrijednost ovisnih varijabli. Reprezentacija modela je u vidu običnog linearnog pravca s nagibom i odsječkom. Bolje je interpretabilnosti od metode neuronske mreže, jer za razliku od nje, vidljivije je koja ulazna varijabla utječe na promjenu izlazne varijable.

Regularizacija je bilo kakva modifikacija u algoritmu učenja koja smanjuje generalizacijsku grešku u testnom skupu podataka. Glavna namjena joj je smanjiti kompleksnost modela strojnog učenja kako bi isti učio preko jednostavnije funkcije. Razlikujemo više

tipova regularizacija, a počet ćemo s takozvanom L1-regularizacijom. Poznata i pod nazivom LASSO (eng. Least Absolute Shrinkage and Selection Operator) dodaje kazneni izraz na funkciju gubitka. Vidljivo u sljedećoj jednakosti:

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{r} X_{i,j}\theta_j)^2 + \lambda \sum_{j=1}^{r} |\theta_j|$$

Radi se o desnom članu, gdje je λ ključ za namještanje modela kako bi se reducirala pojava preciznih prognoza za trening skup, ali ne i za nove podatke. Ta pojava je poznata pod engleskim nazivom overfitting. Prilikom deriviranja funkcije gubitka, L1-regularizacija pokušava procijeniti vrijednost koja bi trebala biti medijan distribucije skupa podataka. Primjerice, uzmimo proizvoljnu vrijednost iz promatranog skupa podataka koji se nalazi na nekom pravcu. Cilj je, kada računamo gubitak, da su vrijednosti s jedne strane dotične proizvoljne vrijednosti manje, a s druge strane veće kod računanja funkcije gubitka.

Linearna regresija s blokovima i L1 regularizacijom

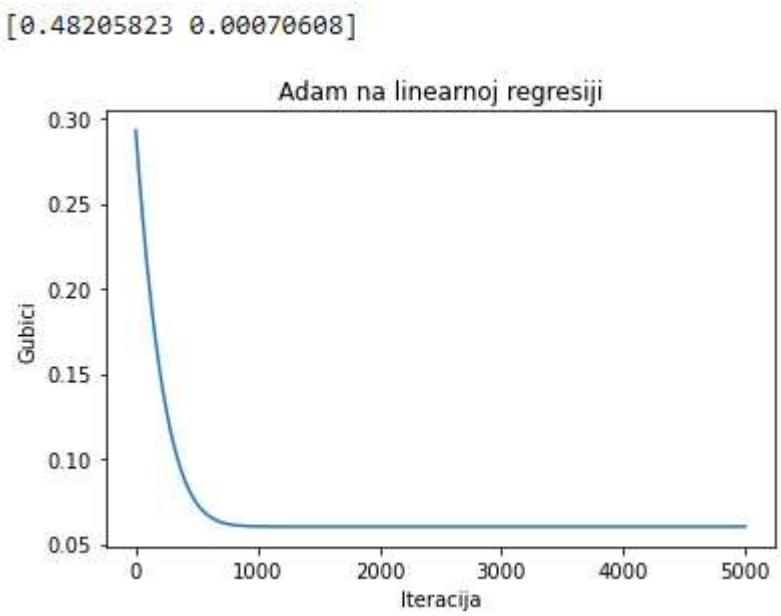
Implementacija same metode linearne regresije je priložena na idućoj slici:

```
def linearna_regresija(X, y, koraci = 10000):
    X = np.concatenate((np.ones((X.shape[0], 1)), X), axis=1)
    #print(X)
    theta = np.zeros(X.shape[1])
    m = np.zeros(len(theta))
    v=np.zeros(len(theta))
    gubici = []
    thete = []
    for i in range(koraci):
        y_prognoza = np.dot(X, theta)
        gubitak = np.mean((y_prognoza - y) ** 2) + 0.1 * np.sum(np.abs(theta))
        gubici.append(gubitak)
        grad = (2 * np.dot(X.T, y_prognoza - y) + 0.1 * np.sign(theta))/ X.shape[0]
        theta, m, v = adam(theta, grad, i, m, v)
        if i == koraci -1:
            print(theta)
        thete.append(theta)

    return thete, gubici
```

Slika 3.1: Implementacija metode linearne regresije s L1 regularizacijom

Koristimo Adama implementiranog u drugom poglavlju s nekim preinakama kako bi bio kompatibilan s metodom. Slijedi prikaz Adam metode na linearnoj regresiji, ali bez L1

regularizacije. Na samoj slici je iznad vidljiva vrijednost posljednjih vrijednosti parametra θ.



Slika 3.2: Adam na linearnoj regresiji bez L1 regularizacije

Težine, odnosno odsječak i nagib u ovom slučaju su dobiveni pozivima već ugrađenih funkcija iz stats biblioteke radi usporedbe s našom implemenacijom te je njihova vrijednost jednaka sljedećem ispisu:

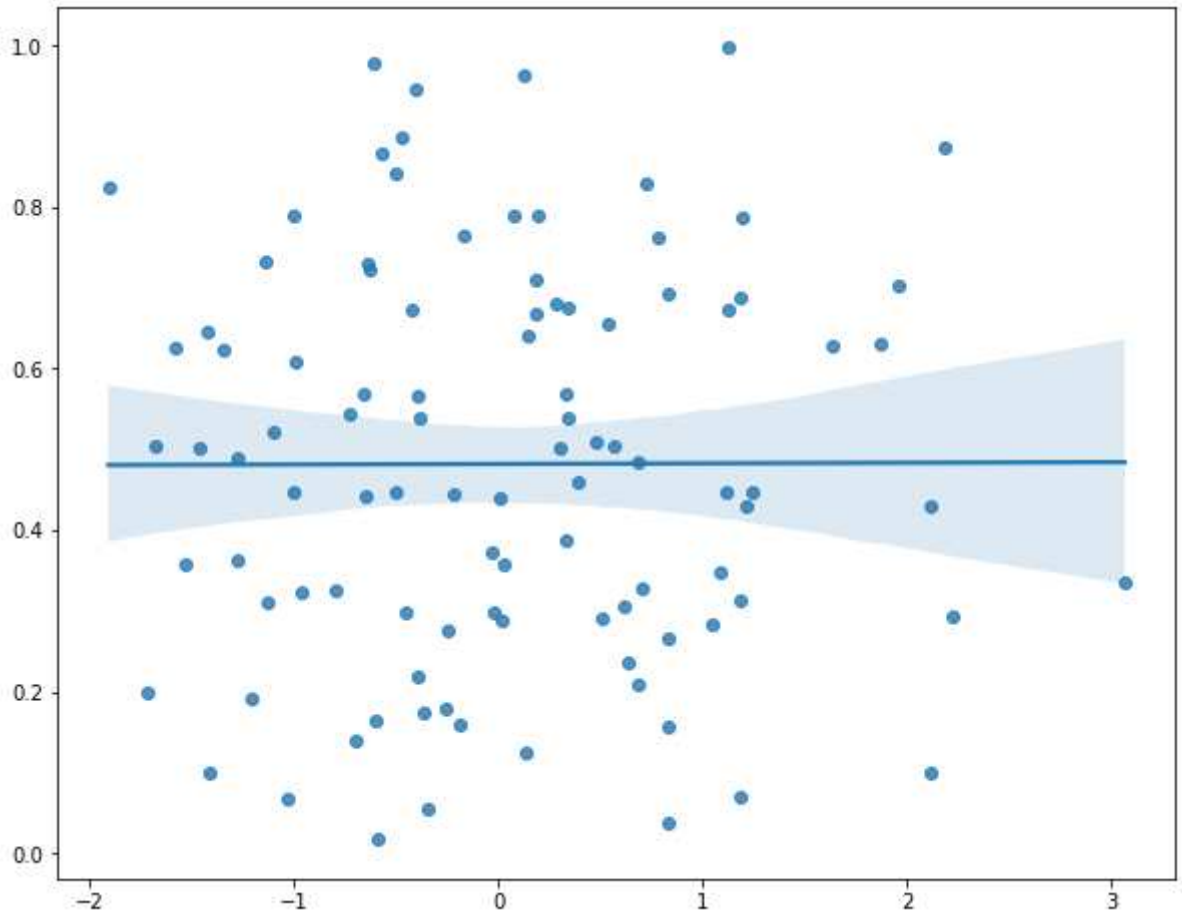
```
from scipy import stats
import seaborn as sns
from matplotlib import pyplot


print("Odsječak: " ,stats.mstats.linregress(X,y).intercept)
print("Nagib: ", stats.mstats.linregress(X,y).slope)
```

```
Odsječak:  0.482058231469521
Nagib:  0.0007060775534202627
```

Slika 3.3: Vrijednost težina pomoću gornjih poziva funkcija

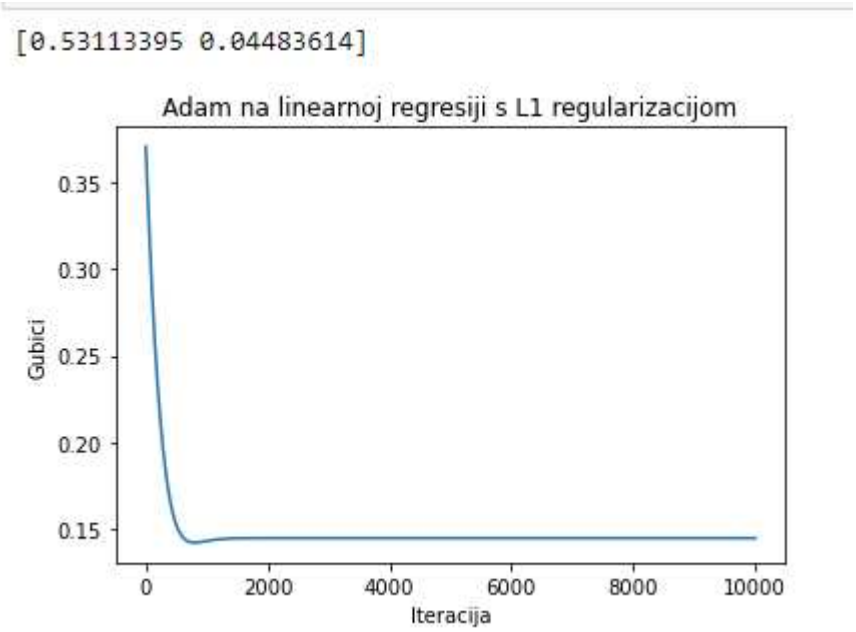
Vidimo da se radi o gotovo sto postotnom poklapanju. Većim brojem iteracija bi se vrijednosti i više podudarale.

Naposljetku, vizualiziramo linearnu regresiju s danim parametrima pomoću seaborn biblioteke:



Slika 3.4: Vrizualizacija pomoću sns.reg

Slijedi prikaz Adama s L1-regularizacijom gdje su u slici iznad ponovno navedene posljednje vrijednosi θ radi usporedbe s pozivima prethodno navedenih funkcija.



Slika 3.5: Izgled linearne regresije s L1 preko Adama

```
print("Odsječak " ,stats.mstats.linregress(X,y).intercept)
print("Nagib: ", stats.mstats.linregress(X,y).slope)
```

```
Odsječak  0.5316483442528325
Nagib:  0.04537984918883379
```

Slika 3.6: Izgled linearne regresije s L1 preko Adama

Primjećujemo manje odstupanje u parametrima budući da funkcijski pozivi nisu uračunali L1 regularizaciju koja služi za smanjenje overfitting pojave. Stabilizacija ponovno nastupa oko 1000. iteracije, ali uz zamjetan pad u gubicima prije iste.

Linearna regresija s blokovima i L2 regularizacijom

Uvodno o L2 regularizaciji

Poznata i kao Ridge regularizacija, kao i L1 dodaje kazneni izraz funkciji gubitka, ali se razlikuje u dotičnom kaznenom izrazu, naime, dodajemo kvadratnu vrijednost težina u cijeli gubitak. Vidljivo u sljedećoj jednakosti:

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{r} X_{i,j}\theta_j)^2 + \lambda \sum_{j=1}^{r} \theta_j^2$$

Kada računamo funkciju gubitka kod L2-regularizacije u svakom koraku, pokušavamo minimizirati gubitak tako da ga odbijamo od prosjeka distribucije podataka. Drugim riječima, L2-regularizacija postavlja ograničenja na težine tako da lambda igra ulogu kaznenog izraza pomoću kojeg se vrši regularizacija težina koje daju velike vrijednosti te zbog kojih su odstupanja modela veća. Regularizacija smanjuje te koeficijente i time reducira složenost modela te multikolinearnost, odnosno statistički koncept u kojem su nekoliko nezavisnih varijabli u korelaciji. Više se može pročitati u [4]. Kod je sličan kodu linearne regresije s L1-regularizacijom, izmjene se vrše samo u računanju gubitka i derivacije izraza gubitka.

```
def linearna_regresija(X, y, koraci = 10000):
    X = np.concatenate((np.ones((X.shape[0], 1)), X), axis=1)
    #print(X)
    theta = np.zeros(X.shape[1])
    m = np.zeros(len(theta))
    v=np.zeros(len(theta))
    gubici = []
    thete = []
    for i in range(koraci):
        y_prognoza = np.dot(X, theta)
        gubitak = np.mean((y_prognoza - y) ** 2) + 0.1  * np.sum(np.square(theta))
        gubici.append(gubitak)
        grad = (2 * np.dot(X.T, y_prognoza - y) + 2* 0.1 * theta)/ X.shape[0]
        theta, m, v = adam(theta, grad, i, m, v)
        if i == koraci -1:
            print(theta)
        thete.append(theta)

    return thete, gubici
```

Slika 3.7: Linearna s L2-regularizacijom

Razlike između L1 i L2 regularizacije

Naveli smo već razliku u pristupu smanjivanja funkcije gubitka kod pojedinih regularizacija, odnosno medijan na skupu podataka te prosjek na skupu podataka kod L1 i L2 respektivno. L1-regularizacija eliminira težine (značajke modela) koje nisu bitne za model ispitivanja. Ovo je prilično korisno kada se radi o velikoj količini dotičnih značajki. Ostale nabrojane razlike su preuzete s [7] te ćemo poneke pokušati vizualizirati i usporediti.
Karakterističnosti L1-regularizacije:

- ima raštrkana rješenja
- ima više rješenja
- ima ugrađen odabir značajki (komponente vektora $\theta \in \mathbb{R}^d$ nazivamo značajke modela)
- robustan je u odnosu na stršeće vrijedonosti (outliere)
- generira model koji je jednostavan i interpretabilan, međutim, ne može naučiti kompleksne obrasce

Karakterističnosti L2-regularizacije:

- nema raštrkana rješenja
- ima jedinstveno rješenje
- nema ugrađen odabir značajki
- nije robustan na prisustvo stršećih vrijedonosti (outlier-a)
- generira model koji može naučiti kompleksne obrasce
- daje bolje predviđanje kada je izlazna varijabla funkcija svih ulaznih značajki (težina)

U suštini L1-regularizacija potiče da su vrijednosti težina 0, dok L2 potiče konvergenciju istih prema 0, ali nikada nije jednaka 0.

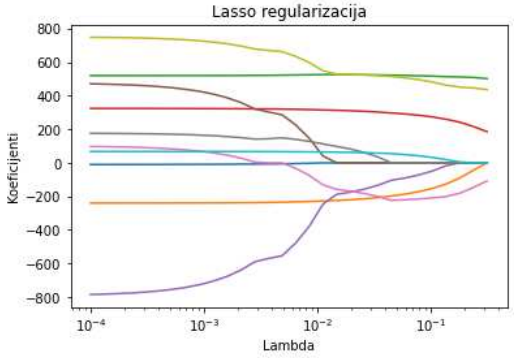
Grafički prikaz L1 i L2

U Lasso regularizaciji općenito nekoliko koeficijenata poprime vrijednost 0 kako λ raste, iz čega dobijemo raštrkana rješenja. Kod Ridge regularizacije to nije slučaj. U L2-regularizaciji također zamjećujemo glađe opadanje u veličini koeficijenata kako λ raste. Rezime usporedbe je da Lasso proizvodi raštrkana rješenja s nekoliko bitnih značajki dok Ridge daje manje raštrkana rješenja koja sadržavaju sve značajke s manjim koeficijentima. Izbor metode regularizacije ovisi o problemu.

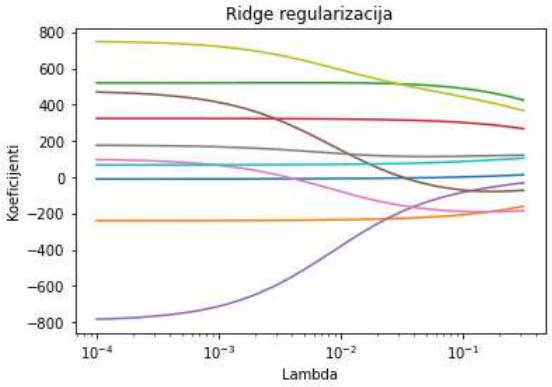
Regularizacija elastične mreže

Tip regularizacije koji koristi i L1 i L2 regularizacije kako bi izbacio optimalan rezultat. Funkcija gubitka je dana sljedećim izrazom:

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{r} X_{i,j}\theta_j)^2 + \lambda(\alpha \sum_{j=1}^{r} |\theta_j| + (\frac{1-\alpha}{2}) \sum_{j=1}^{r} \theta_j^2)$$



Slika 3.8: Lasso regularizacija



Slika 3.9: Ridge regularizacija

Također je moguće namještati λ, no ovdje je i dodan parametar α gdje je za vrijednost 1 ova regularizacija istovjetna Lasso-u, a za 0 Ridgeu. Koristi se kada postoji mnogo beskorisnih varijabli koje bi bilo poželjno ukloniti te mnogo korisnih koje treba zadržati. Također je bolji od obje regularizacije kada proučava korelirane varijable.

Linearna regresija bez regularizacija na ostalim metodama

Usporedit ćemo izmjene težina na RMSProp-u, Momentu i Nesterovljevom akceleriranom gradijentu. Izvršene su na istim testnim podacima. Ponovno pozivamo ugrađene funkcije kako bismo znali vrijednosti θ.

```
print("Odsječak " ,stats.mstats.linregress(X,y).intercept)
print("Nagib: ", stats.mstats.linregress(X,y).slope)
```

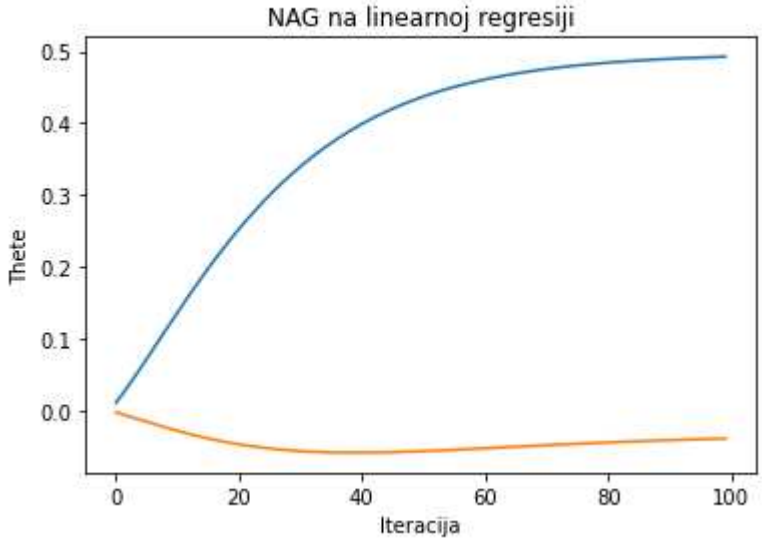
```
Odsječak  0.5105822804812933
Nagib:  0.004210264251103162
```

Slika 3.10: Težine za ostale metode

```
[ 0.50148957 -0.01210644]
```

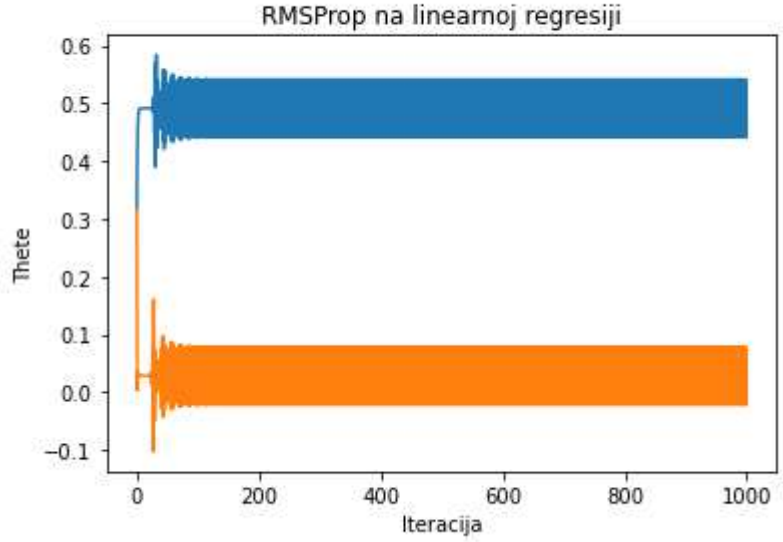
Slika 3.11: Moment na linearnoj regresiji

```
[ 0.49268611 -0.03878845]
```



Slika 3.12: NAG na linearnoj regresiji

```
[ 0.4411457  -0.02146825]
```



Slika 3.13: RMSProp na linearnoj regresiji

Primjećujemo veća odstupanja kod moment i NAG metoda, gdje je moment i veći prijestupnik, te nijedna ne teži dobro prema drugoj dimenziji parametra u odabranom broju iteracija. RMSProp je stabilniji te na kraju nakon minornih, ali stalnih oscilacija teži k približnim traženim vrijednostima. Debljina linija je zbog broja iteracija.

Poglavlje 4

Autograd i Jax

4.1 Autograd

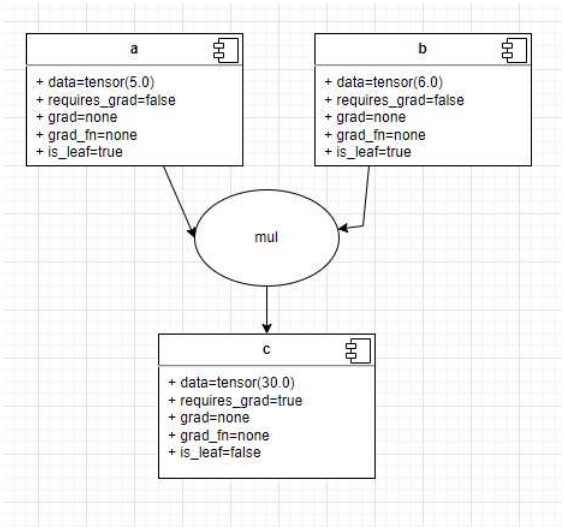
Uvod-općenito

Za ovo poglavlje referentni su Internet izvori [2] i [15]. **Tenzor** je višedimenzionalno polje s uniformnim tipom podataka. Zapravo se radi o poopćenju vektora i matrica u višim dimenzijama.

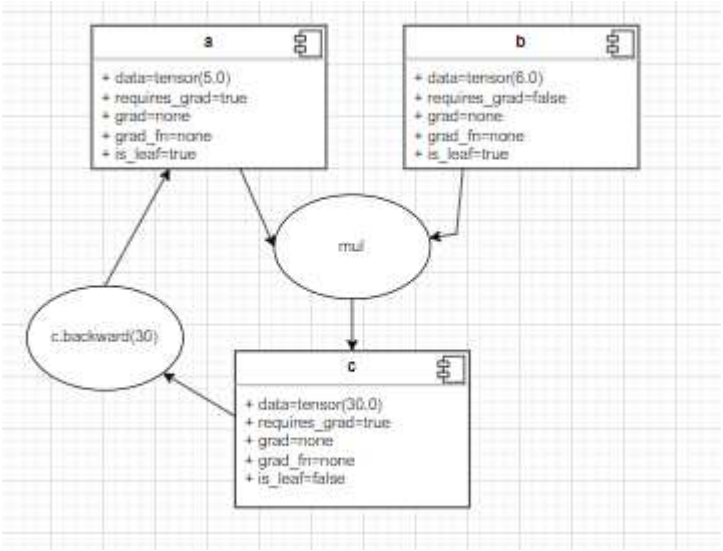
PyTorch je jedna od najkorišetnijih biblioteka Pythona za duboko učenje. Budući da su nerijetko nužni teški izračuni za pojedine probleme strojnog učenja, razvijen je naslovni paket Autograd unutar PyTorcha koji se koristi za automatsko deriviranje na svim operacijama u tenzorima. To je klasa koja računa Jacobijev produkt matrica pri čemu pamti graf svih operacija izvršenih na tenzoru ispitivanja. Stvara aciklički graf kojeg nazivamo dinamički graf izračuna (eng. dynamic computational graph - DCG). Sam graf je izmjenjiv, te su mu listovi ulazni tenzori, a korijeni su izlaz. Gradijenti se računaju po principu lančanog pravila od korijena do lista računajući svaki gradijent tim putem. Na slici je prikazan dinamički graf izračuna te komponente kojima su atributi redom:

- `data`: u a se nalazi tenzor kao skalar u float vrijednosti 5.0
- `requires_grad`: ukoliko je vrijednost na true, stvori se reverzni graf koji pamti unazad sve operacije na tenzoru kako bi izračunao gradijent
- `grad_fn`: reverzna funkcija koja računa gradijent
- `grad`: sadrži vrijednost gradijenta. Postavljen na none ako je `requires_grad` false ili dok nije neka reverzna funkcija pozvana

- `is_leaf`: true ako je pozvana inicijalizacijom tenzora, `requires_grad` prethodnika je postavljen na false ili nakon .detach() poziva funkcije. Inače true.



Slika 4.1: DCG-bez gradijenta: izrađeno u draw.io



Slika 4.2: DCG-s gradijentom: izrađeno u draw.io

U c.Backwardu() sprema se trenutni kontekst, tj., tenzori koji su proživjeli neke transformacije unutar programa. Sadrži i atribut `next_functions` u kojem je spremljen tuple koji pamti za koji je tenzor nužna akumulacija gradijenta, odnosno pamti zastavice `requires_grad`. Akumulacija gradijenta mijenja zastavicu grad zbrajajući prethodne izračunate vrijednosti iz prijašnjih operacija te ju postavlja na taj zbroj. Tu radnju nazivamo "backward-pass".

Matematička podloga

Teorem 4.1.1 (Lančano pravilo)**.** *Neka su $f : I \to \mathbb{R}$, $g : J \to \mathbb{R}$ i neka je $f(I) \subseteq J$, tj. kompozicija $g \circ f : I \to \mathbb{R}$ je dobro definirana na I. Ako je funkcija f diferencijabilna u točki $c \in I$ i funkcija g diferencijabilna u točki $d = f(c) \in J$, onda je kompozicija $g \circ f$ diferencijabilna u c i vrijedi $(g \circ f)'(c) = g'(d)f'(c)$.*

Kako je prethodno navedeno, Autograd je zapravo klasa koja koristi Jacobijev produkt matrica za prikaz rezultata. Ako koristimo vektor $X = [x_1, x_2, ..., x_n]$ kako bi izračunali vektor u ovisnosti funkcije f, tada je Jacobijeva matrica zapravo matrica koja sadrži sve moguće kombinacije parcijalnih derivacija vektora $f(X)$.

$$J = \begin{bmatrix} \frac{df_1}{dx_1} & \cdots & \frac{df_1}{dx_n} \\ \vdots & \ddots & \vdots \\ \frac{df_m}{dx_1} & \cdots & \frac{df_m}{dx_n} \end{bmatrix}$$

Neka je $v = (\frac{dl}{dy_1}, \ldots, \frac{dl}{dy_n})$ gradijent skalara gubitka u odnosu na y=f(X).

$$Jv = \begin{bmatrix} \frac{dl}{dx_1} \\ \vdots \\ \frac{dl}{dx_n} \end{bmatrix}$$

Gornji umnožak daje gradijent funkcije gubitka.

Napomena 4.1.2. *Autograd ne mora biti upoznat sa strukturom programa u kojem je pozvan, točnije, ne treba znati o if-else naredbama, petljama ni rekurzijama koje su pozvane radi izvršenja operacija na tenzoru ispitivanja, odnosno inputu. Kako bi se izračunao gradijent inputa potrebno je samo znati niz transformacija obavljenih operacijama tijekom izvršavanja programa. Autograd prati sve relevantne operacije za svaki funkcijski poziv odvojeno, stoga nema potrebe pratiti operacije toka programa Pythona što omogućuje puno lakšu implementaciju.*

Mogućnosti Autograda

Primjer 1

Prikazat ćemo primjer po uzoru na primjer iz referentne stranice, kako bismo uvidjeli da Autograd zadovoljava prethodnu napomenu.

```
import autograd.numpy as np
from autograd import grad
import math
def cos(x):
    cos_aproks = 0
    for i in range(5):
        koef = (-1)**i
        cos_aproks += ( koef ) * ( (x**(2*i)/(math.factorial(2*i)) ))
    return cos_aproks
grad_cos=grad(cos)
print ("Gradijent od cos(pi):", grad_cos(np.pi))
```

```
Gradijent od cos(pi): 0.07522061590362306
```

Slika 4.3: Gradijent u aproksimaciji kosinusa

Uočimo kako je funkcija grad pozvana samo na kraju.

Primjer 2 - Dekorator primitive

Dekorator primitive koristimo u slučaju funkcija koje Autograd ne podržava, te se zapravo radi o svojevrsnom proširenju same klase. Autograd tada zamišlja samu funkciju kao crnu kutiju. Svaka funkcija koju Python može izvršiti, makar je iz biblioteke drugog programskog jezika, se može dodati na ovaj način. Ovaj primjer će također biti sličan primjeru iz [15], ali ne i identičan. Implementirat ćemo sigmoidalnu funkciju na sličan način.

```
import autograd.numpy as np
from autograd.extend import primitive, defvjp
from autograd import grad

@primitive
def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoid_vjp(ans, x):  #vraća vektor Jacobiana produkta; identično tutorialu
    x_shape = x.shape
    return lambda g: np.full(x_shape, g) * np.exp(x - np.full(x_shape, ans))

defvjp(sigmoid, sigmoid_vjp)
def primjer(y):
    z = y**3
    pom = sigmoid(z)
    return np.sum(pom)

grad_sig = grad(primjer)
print ("Gradijent: ", grad_sig(np.array([2.8, 3.2, 1e-10])))
```

```
Gradijent:  [2.95648243e+10 1.92348282e+15 1.81959198e-20]
```

Slika 4.4: Gradijent u kompoziciji sa sigmoidalnom funkcijom

Primjer 3 - Adam

```
def mse(theta, x, y):
    y_pred = linearna_regresija(theta, x)
    return np.mean((y_pred - y) ** 2)

g = grad(mse)

for i in range(100):
    grad_theta = g(theta, x, y)

    theta = adam(grad_theta, theta)
```

Slika 4.5: Gradijent pomoću Autograda

Autograd ima mogućnost automatski derivirati funkciju zbog čega nema potrebe definirati derivaciju ciljne funkcije kao zasebnu funkciju ili na neki drugi način. Također u svom libraryju sadrži i implementiranog samog Adam optimizatora što dodatno ubrzava proces primjene istog na metode strojnog učenja.

Primjer 4 - Autograd direktno korištenje

Po uzoru na primjer s web-stranice [9] modificiran je kod tako da koristi metodu Adam. Tu vidimo direktnu primjenu autograda s ugrađenim funkcijama kako računa linearnu regresiju.

```
learning_rate = 0.01
n_iters = 100

optimizer = optim.Adam([w], lr=learning_rate)

for epoch in range(n_iters):
    # predviđanje = forward pass
    y_pred = forward(X)

    # loss
    l = loss(Y, y_pred)

    # računanje gradijenata = backward pass
    l.backward()

    # ažuriranje težina
    optimizer.step()

    # postavljanje gradijenata na 0 nakon ažuriranja
    optimizer.zero_grad()

    if epoch % 10 == 0:
        print(f'epoha {epoch+1}: w = {w.item():.3f}, gubitak = {l.item():.8f}')

print(f'Predviđanje nakon treninga: f(5) = {forward(5).item():.3f}')
```

```
Predviđanje prije treninga: f(5) = 0.000
epoha 1: w = 0.010, gubitak = 30.00000000
epoha 11: w = 0.110, gubitak = 27.07978249
epoha 21: w = 0.209, gubitak = 24.33392334
epoha 31: w = 0.306, gubitak = 21.77764511
epoha 41: w = 0.400, gubitak = 19.41691971
epoha 51: w = 0.493, gubitak = 17.25031281
epoha 61: w = 0.582, gubitak = 15.27167797
epoha 71: w = 0.668, gubitak = 13.47233200
epoha 81: w = 0.752, gubitak = 11.84241581
epoha 91: w = 0.832, gubitak = 10.37159348
Predviđanje nakon treninga: f(5) = 4.508
```

Slika 4.6: Gradijent direktno pomoću Autograda

4.2 Jax

Glavna referenca za ovo potpoglavlje je web stranica [10]. Jax je objedinjenje već spomenutog Autograda i XLA, optimizirajućeg kompajlera za strojno učenje, kako bi omogućio strojno učenje visokih performansi. Budući da sadrži Autograd pokriva sve njegove navedene funkcionalnosti. Novina u odnosu na Autograd je upravo XLA pomoću kojeg je omogućeno kompiliranje na grafičkim karticama i tenzorskoj procesuirajućoj jedinici. Također omogućuje kompilaciju autorskih Python funkcija u XLA-optimiziranim jezgrama koristeći funkciju API, jit.

NumPy

Možemo zamisliti JAX kao diferencijabilni NumPy koji se vrti na grafičkim akceleratorima. Programi koji koriste NumPy mogu se pokretati u JAX-u ako samo zamijenimo np za jnp, međutim postoje neke razlike. Jedna od tehničkih razlika je kada pri pozivu JAX programa, odgovarajuća operacija bude poslana u akcelerator i računa se asinkrono kadgod je to moguće. Ukoliko je rezultat odmah potreban može doći do blokade kod izvršavanja Python programa. Najbitnija razlika, koja je u suštini zapravo i korijen svih ostalih razlika, jest ta da je JAX zamišljen u funkcionalnom, a ne proceduralnom programiranju, budući da je tako lakše vršiti transformacije za koje je predviđen. Programe koji koriste JAX je preporučljivo kompilirati prije pokretanja ostatka koda s transformacijom jax.jit pa kod postaje efikasan.

grad

Jedno od osnovnih svojstava JAX-a je da dozvoljava transformacije funkcija, a u kontekstu ovog rada, najbitnija transformacija je jax.grad, koju možemo smatrati analogonom gradijentu, slično kao i u autogradu. Njoj je ulazni parametar neka numerička funkcija, dok je izlaz nova funkcija pomoću koje se računa gradijent dotične ulazne funkcije. Po uzoru na primjere iz navedene reference ovog potpoglavlja će biti napisani slični, ali ne i identični primjeri koji će demonstrirati istu funkcionalnost. Za prvi primjer uzet ćemo funkciju gubitka poznatu kao eng. root mean squared error, odnosno korijen kvadrata srednje vrijednosti.

```
import jax.numpy as jnp
from jax import grad, jit, vmap

def mean_greska_korijen(y_stvarno, y_predvid):
    return jnp.sqrt(jnp.mean(jnp.square(y_stvarno - y_predvid)))

y_stvarno = jnp.asarray([1., 2., 3., 4.])
y_predvid = jnp.asarray([3.1, 4.1, 3.1, 4.1])

rmse_grad = grad(mean_greska_korijen, argnums=1)(y_stvarno, y_predvid)
print(rmse_grad)
```

```
[0.35315323 0.35315323 0.01681681 0.01681681]
```

Slika 4.7: Gradijent pomoću Jaxa

Prirodno se nameće pitanje je li potrebno pisati funkcije s ogromnim listama argumenata. No, JAX se i za to pobrinuo tako da ima ugrađenu mogućnost da spaja polja u strukturu podataka znanu kao **pytree**.

Pytree

Definicija je preuzeta iz referentne stranice na početku potpoglavlja. Radi se o spremniku elemenata naziva listovi ili se sastoji od još pytree-ova. Spremnici mogu sadržavati liste, tuple-ove i rječnike. Listovi su bilo koji element koji nije pytree, primjerice polje. Dakle, pytree-ovi su ugniježđeni standardni ili korisnički napravljeni Python spremnici. Ne moraju se poklapati tipovi spremnika u slučaju ugnježđavanja. Osnovni list također smatramo pytree-om. U strojnom učenju pronalazimo ih u parametrima modela, unos skupa podataka i u agentima pojačanog učenja(eng. reinforcement learning). jax.lax.scan() je jedna od mnogobrojnih funkcija JAX-a koje koriste pytree-ove polja te vrše nad njima svoje transformacije funkcija koje su i ulaz istih. Ovu strukturu možemo proširivati, korisnički prilagoditi i interno ju izmjenjivati što omogućuje i veću fleksibilnost radi rukovanja s njom. Najkorištenija funkcija je jax.tree_map koja je analogon Pythonovog map. Navedenu možemo i koristiti za transponiranje, primjerice ako imamo stablo listi, možemo ga prebaciti u listu stabala. Na slici s početka iduće stranice smo napravili jedan pytree od 3 lista gdje je svaki list polje s 3, 2 i 4 elementa respektivno.

```
podaci = [
    [1, 3, 5],
    [1, 4],
    [1, 4, 9, 16]
]

jax.tree_map(lambda x: x*4, podaci)
```

```
[[4, 12, 20], [4, 16], [4, 16, 36, 64]]
```

Slika 4.8: Primjer poziva jax.tree_map

Automatsko diferenciranje

Kao i u autogradu, bazira se na backpropagation pristupu računanja diferencijala. Promotrit ćemo nekoliko primjera koji će ukazati na mogućnosti JAX-a i kako je to primjenjivo na našu metodu linearne regresije s Adamom.

Primjer 1- Derivacije višeg reda

Jednostavan primjer koji pokazuje da je samo gomilanje poziva funkcije jax.grad() dovoljno za željeni red derivacije.

```
import jax

f = lambda x: x**5 + 16*x**4 + 2*x**2 + 3

d = jax.grad(f)
d2 = jax.grad(d)
d3 = jax.grad(d2)
d4 = jax.grad(d3)
d5 = jax.grad(d4)
print(d(1.))
print(d2(1.))
print(d3(1.))
print(d4(1.))
print(d5(1.))
```

```
73.0
216.0
444.0
504.0
120.0
```

Slika 4.9: Primjer deriviranja do reda 5 s 1 varijablom

Za deriviranje funkcije višeg reda, primjerice za 2. reda nam je potreban Hessian, kojeg možemo prikazati kao Jacobian prvog gradijenta. Tu JAX daje 2 funkcije jax.jacfwd() i

jax.jacrev() koje daju rješenje. Dovoljno je definirati Hessian s jax.jacfwd(jax.grad(f)) ili jax.jacrev(jax.grad(f)).

Primjer 2- Izravni procjenitelj

Radi se o zaobilaznom načinu definiranja gradijenta funkcije koja obično nije diferencijabilna. Kada je dotična dio veće funkcije od koje želimo gradijent, ponašamo se u backpropagationu da je ista identiteta. Na referentnoj stranici ćemo po uzoru na funkciju $|x|$ u ovom primjeru implementirati maksimum između x i y te koristimo lax.stop_gradient.

```
import jax.numpy as jnp
from jax import grad, vmap

def izravni_procjenitelj_max(x, y, gubitak):
    maxi = jnp.maximum(x, y)
    grad_x = gubitak * (x >= y)
    grad_y = gubitak * (y >= x)
    maxi + lax.stop_gradient(grad_x + grad_y - maxi)
    return maxi, grad_x, grad_y

x = jnp.array([1., 7., 9.])
y = jnp.array([5., 3., 1.])
gubitak = jnp.array([0.5, 0.4, 0.3])
izlaz, grad_x, grad_y = izravni_procjenitelj_max(x, y, gubitak)

print("Izlaz:", izlaz)
```

```
Izlaz: [5. 7. 9.]
```

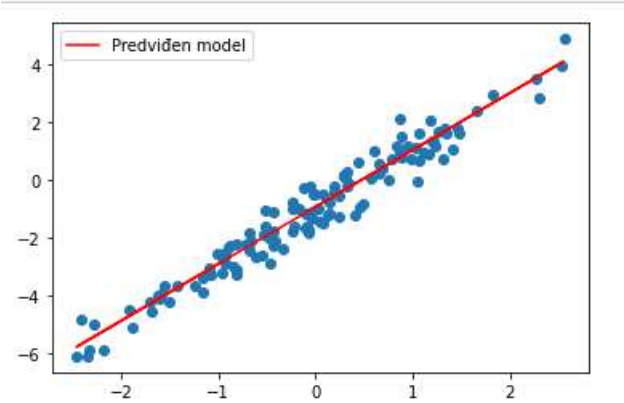
Slika 4.10: Primjer izravnog procjenitelja za max

Primjer 3- Izračunavanje stanja

Na referentnoj stranici je dan primjer linearne regresije sa SDG optimizatorom te upute u komentarima kako bismo mogli dobiti izračunavanje s Adam optimizatorom. To ćemo napraviti u ovom primjeru. Naime, treba samo izmijeniti update funkciju. Samim time dobijemo istu sliku kao i sa SDG koji je dovoljan za optimizaciju ovog jednostavnog modela. Motivacija ovog primjera je ta što neke JAX transformacije nameću ograničenja na funkcije koje transformiraju. Transformirane funkcije ne smiju imati nuspojave budući da se one izvrše samo jednom te ne više pri idućim izvođenjima programa. Naslovna promjena stanja je jedna od dotičnih nuspojava. Kako bismo ju izbjegli, pokušavamo ažurirati parametre modela i stanja optimizatora preko funkcijskog programiranja. Za puno više parametara se brinu neke JAX biblioteke.

```
LEARNING_RATE = 0.005
opt_init, opt_update, get_params = optimizers.adam(step_size=LEARNING_RATE)


@jit
def update(i, opt_state, x, y):
  model_params = get_params(opt_state)
  g = grad(loss)(model_params, x, y)
  return opt_update(i, g, opt_state)
```

Slika 4.11: Primjer izračunavanja stanja s Adamom



Slika 4.12: Predviđeni model

Još neke funkcionalnosti

- Optimizacija višeg reda - diferenciranje tijekom ažuriranja gradijenata
- Gradijent po uzorku u batchu - radi efikasnosti izračunavanja i smanjenja varijanci ponekad je neophodno znati gradijent samo pojedinog uzorka
- Stopirajući gradijent - kada želimo dodatnu kontrolu te želimo ukloniti računanje nekih gradijenata u određenom skupu grafa izračunavanja

Paralelizam

JAX podržava paralelizam preko vektorizacije i na uređajima. Za vektorizaciju koristimo funkciju jax.vmap, a za uređaje jax.pmap kako bi tranfsormirale funkciju namijenjenu za 1 uređaj u funkciju koja operira paralelno na više uređaja. Automatska vektorizacija

omogućuje jednostavnu vektorizaciju funkcije bez da se korisnik zamara s indeksima i osima te ostalim dijelovima ulaznih vrijednosti. Paralelizam na uređajima počinje tako da su svi ulazni parametri na domaćinu (CPU). Pomoću pmap-a šaljemo ih ostalim uređajima nakon poziva funkcije update() te tada svaka kopija ostaje na svom uređaju. Tijekom izvršavanja update() funkcije, potrebno je nekako iskombinirati gradijente izračunate na uređajima, inače bi ažuriranja bila drukčija na svakom. U tom slučaju se koristi funkcija jax.lax.pmean koja daje prosječni gradijent bloka. Podaci istraživanja (eng. data) se ne mijenjaju te stalno ostaju na CPU domaćinu te preko pipeline pristupa se svakim novim pozivom update() šalju ostalim uređajima.

Poglavlje 5

Rekonstrukcija slike kao problem upotpunjavanja matrice

Referentni članak po kojem je bazirano ovo poglavlje je [16].

Motivacija i opis problema

Rekonstrukcija matrice ima važnu ulogu u strojnom učenju i rudarenju podataka. U ovom primjeru radi se o postupku rekonstrukcije u kojem se oštećeni ili nedostajući dijelovi neke slike popunjavaju kako bismo dobili cjelovitu sliku. Koristimo matrice te matematičke izračune u ovisnosti na okolne piksele kako bismo procijenili kakvog su zapravo izgleda ti nedostajući dijelovi. Koristit ćemo algoritam **SVD** (eng. singular value decomposition) u kombinaciji s **Huberovom** funkcijom gubitka kako bismo upotpunili rijetku matricu koja služi kao prikaz objekta promatranja, odnosno, oštećene slike. Cilj algoritma iz članka je dodati i Gaussove šumove i šumove izazvane rijetkim outlierima.
SVD je dekompozicija matrice X takva da faktoriziramo X u produkt 3 matrice, to jest, $X = UDV^T$ gdje su stupci od U i V ortonormirani i matrica D je dijagonalna s pozitivnim realnim vrijednostima.
Huberova funkcija gubitka je definirana kao

$$f_H(x) = \begin{cases} x^2, & |x| \leq c \\ c(2|x| - c), & x > c, \end{cases}$$

gdje je $c > 0$ parametar. Kada parametar c teži 0 funkcija gubitka je ekvivalentna l_1 gubitku, a kada teži ∞, onda je ekvivalentna l_2 funkciji gubitka.

```
def huber(x, c):
    if abs(x) <= c:
        return x**2
    else:
        return c(2 * abs(x) - c)
```

Slika 5.1: Huberova funkcija gubitka

Model koji koristimo za rekonstrukciju slike glasi

$$\min_{U^TU=I_d,V} \|W \odot (M - UV)\|_H + \lambda\|V\|_*, \tag{5.1}$$

gdje je s $\odot$ označen Hadamardov produkt matrica:

$$(A \odot B)_{i,j} = a_{i,j}b_{i,j},$$

a sa $*$ je označena norma definirana sa

$$\|A\|_* = \sum_{i=1}^{r} \sigma_i(A),$$

gdje je sa $\sigma_i(A)$ označena i-ta singularna vrijednost matrice A. Matrica W je indikatorska matrica koja na mjestu (i, j) ima vrijednost 1 ako reducirana matrica na mjestu (i, j) ima poznatu vrijednost, inače ima vrijednost 0. U gornjem modelu s r je označen rang reducirane matrice te vrijedi $d \geq r$. Rješavanje ovog problema će nam olakšati ADMM algoritam (eng. alternating direction method of mulitpliers). U algoritmu ćemo koristiti Frobeniusovu matričnu normu koja služi kao mjera za „veličinu" definiranu kao:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{i,j}^2|},$$

gdje je A matrica s m redaka i n stupaca te $|A_{i,j}|$ apsolutna vrijednost elementa u i-tom retku i j-tom stupcu.

Optimizacijski algoritam

U svrhu rješvanja ovog problema uvodimo varijablu L i zapisujemo gornji problem kao

$$\min_{U,V,L} \|W \odot (M - L)\|_H + \lambda\|V\|_*,$$

gdje je

$$L = UV, U^T U = I_d.$$

Uvodimo proširenu Lagrangeovu funkciju kao

$$\mathcal{L}(U, V, L, Y, \mu) = \|W \odot (M - L)\|_H + \lambda\|V\|_* + < Y, L - UV > + \frac{\mu}{2}\|L - UV\|_F^2$$

tako da

$$U^T U = I_d.$$

Y označava Lagrangeov multiplikator, a μ je korigirajući parametar. Za problem ažuriranja varijable U ADMM metoda radi tako da ažurira svaku varijablu fiksirajući reziduale. Sve varijable i Lagrangeove multiplikatore sekvencijalno ažuriramo tijekom svake iteracije. Navodimo kako mijenjamo varijable u $(k + 1)$-oj iteraciji.

Ažuriranje varijable U

Fiksiranjem preostalih varijabli, mijenjamo U rješavajući sljedeći problem:

$$\min_U \|L^k - UV^k + \frac{Y^k}{\mu^k}\|_F^2$$

tako da

$$U^T U = I_d.$$

Pretpostavimo da je SVD od $(L + \frac{Y}{\mu})V^T$ dan sa

$$(L^k + \frac{Y^k}{\mu^k})(V^k)^T = U_t S_t V_t^T.$$

Optimalno rješenje za problem 5.1 je

$$U = U_t V_t^T.$$

Ažuriranje varijable V

Fiksiranjem varijabli L i U, ažuriramo varijablu V rješavajući sljedeći problem

$$\lambda\|V\|_* + \|L^k - U^{k+1}V + \frac{Y^k}{\mu^K}\|_F^2. \tag{5.2}$$

Za

$$U^T U = I_d,$$

gornji problem je ekvivalentan problemu

$$\lambda_1\|V\|_* + \frac{1}{2}\|V - (U^k)^T(L^k + \frac{Y^k}{\mu^k})\|_F^2,$$

gdje je $\lambda_1 = \frac{\lambda}{\mu^k}$. Neka je $P = (U^k)^T(L^k + \frac{Y^k}{\mu^k})$. Pretpostavimo da je SVD od P

$$P = U_p S_p V_p^T.$$

Tada je rješenje problema 5.2 dano jednadžbom

$$V = U_p S_{\lambda_1}(S_p) V_p^t,$$

gdje S_{λ_1} predstavlja operatora ograničavanja singularne vrijednosti (eng. singular value thresholding operator, SVT). Uzevši u obzir i-ti dijagonalni element od S_p, u oznaci σ_i, možemo dobiti i-ti dijagonalni element matrice $S_{\lambda_1}(S_p)$ kao

$$S_{\lambda_1}(\sigma_i) = \max(\sigma_i - \lambda_1, 0).$$

Računanje matrice V se zapravo svodi na problem računanja niza linearnih regresija.

Ažuriranje varijable L

Fiksiranjem varijabli U i V, ažuriramo varijablu L rješavajući sljedeći problem

$$\min_L \|W \odot (M - L)\|_H + \frac{\mu}{2}\|L - U^{k+1}V^{k+1} + \frac{Y^k}{\mu^k}\|_F^2. \tag{5.3}$$

Neka je $L = M - T$. Gornji problem možemo preformulirati kao

$$\min_T \|W \odot (T)\|_H + \frac{\mu}{2}\|T - (M - U^{k+1}V^{k+1} + \frac{Y^k}{\mu^k})\|_F^2.$$

Ulaze od L možemo riješavati odvojeno. Nadalje, imamo

$$\hat{W} \odot T = \hat{W} \odot (M - U^{k+1}V^{k+1} + \frac{Y^k}{\mu^k}),$$

gdje je $\hat{W} = ones(m, n) - W$. Pretpostavimo da je Ω skup indeksa poznatih ulaza. Za $t_{i,j}, (i, j) \in \Omega$ rješavamo sljedeće probleme

$$\min_{t_{i,j}} f_H(t_{i,j}) + \frac{\mu}{2}(t_{i,j} - a_{i,j}), \tag{5.4}$$

gdje $a_{i,j}$ predstavlja (i, j)-ti član matrice $M - U^{k+1}V^{k+1} + \frac{Y^k}{\mu^k}$. Možemo dobiti približno rješenje problema 5.4 pomoću:

$$\begin{cases} t_{i,j} = S_{\frac{2c}{\mu}}(a_{i,j}), & t_{i,j} \geq c \\ t_{i,j} = \frac{\mu a_{i,j}}{2+\mu}, & t_{i,j} < c, \end{cases}$$

gdje je $S_{\frac{2c}{\mu}}(x) = \max(|x| - \frac{2c}{\mu}, 0)sng(x)$ predstavlja apsolutnu vrijednost operatora smanjivanja, a $sng(x)$ je signum funkcija. Možemo ažurirati varijablu L kada je T izračunat.

Ažuriranje Lagrangeovog multiplikatora Y i parametra μ

Lagrangeov multiplikator Y dobivamo jednadžbom

$$Y^{k+1} = Y^k + \mu^k(L^{k+1} - U^{k+1}V^{k+1}),$$

dok parametar $\mu^{k+1} = \min(\rho\mu^k, 10^{12})$, gdje je $\rho = 1.05$ korak u ažuriranju μ. Kriterij zaustavljanja je

$$||L - UV||_F \leq 1e^{-4}.$$

Pseudokod cijelog algoritma

Algoritam 2 HRMC preko ADMM

Ulaz: X ▷ matrica s nedostajućim vrijednostima
Ulaz: r ▷ rang željene matrice niskog ranga
Ulaz: N ▷ maksimalni broj koraka
Ulaz: U, L, V, Y ▷ U i V su matrice niskog ranga za rekonstrukciju, L
Izlaz: U, V, L ▷ rezultirajuće matrice

$N \leftarrow 1000$
$U \leftarrow 0$
$L \leftarrow 0$
$V \leftarrow 0$
$Y \leftarrow 0$
dok $k = 1$ manje od 1000 **radi**
$U \leftarrow U^t V_t^T$ ▷ ažuriranje matrice U po prethodno istaknutom pravilu
$V \leftarrow \lambda \|V\|_* + \|L^k - U^{k+1}V + \frac{Y^k}{\mu^k}\|_F^2$ ▷ ažuriranje matrice V
$T \leftarrow \hat{W} \odot (M - U^{k+1}V^{k+1} + \frac{Y^k}{\mu^k})$
$T \leftarrow \min_{t_{i,j}} f_H(t_{i,j}) + \frac{\mu}{2}(t_{i,j} - a_{i,j})$
$L \leftarrow M - T$ ▷ ažuriranje matrice L
$Y \leftarrow Y^k + \mu^k(L^{k+1} - U^{k+1}V^{k+1})$ ▷ ažuriranje Lagrangeovog multiplikatora
$\mu \leftarrow \min(\rho\mu^k, 10^{12})$ ▷ ažuriranje parametra
ako $\|L - UV\|_F \leq 1e^{-4}$ **tada** prekid ▷ kriterij zaustavljanja
kraj ako
kraj dok

Za prije navedene konstante u optimizaciji uzeli smo da je $\lambda = 0.2$, pomoću kojeg dobivamo λ_1 u ažuriranju matrice V te da je $c = \max(X)$. U ovom dijelu se zapravo radi o alternirajućoj minimizaciji u smislu najmanjih kvadrata po pojedinim faktorima U i V i tome odgovara Adamu. Koristeći Huberovu funkciju gubitka u računanju matrice L, doprinosi se smanjivanju utjecaja ekstremnih vrijednosti u matrici, eng. sparse outliers.

Prilažemo kod kojim se vide izmjene najbitnijih varijabli po pseudokodu te pravilima promjena iz članka.

```
def update_U(L, Y, V, mu):
    tmp = (L + Y / mu) @ V.T
    Ut, St, VtT = np.linalg.svd(tmp, full_matrices=False)
    return Ut @ VtT


def update_V(L, Y, U, mu, _lambda):
    P = U.T @ (L + Y / mu)
    lambda1 = _lambda
    return svt(P, lambda1)


def update_L(X, Y, U, V, W, mu, c):
    # caluclate T
    A = np.zeros_like(X)
    T = np.zeros_like(X)

    A = X - U @ V + Y / mu

    for i in range(T.shape[0]):
        for j in range(T.shape[1]):
            if W[i,j]:
              if T[i, j] > c:
                  T[i, j] = operator_smanjivanja(A[i, j], c, mu)
              else:
                  T[i, j] = (mu * A[i, j]) / (2 + mu)

    return X - T


def update_Y(L, Y, U, V, mu):
    return Y + mu * (L - U @ V)


def update_mu(mu, rho, threshold):
    return min(rho * mu, threshold)


def update_lambda(_lambda, mu):
    return _lambda / mu
```

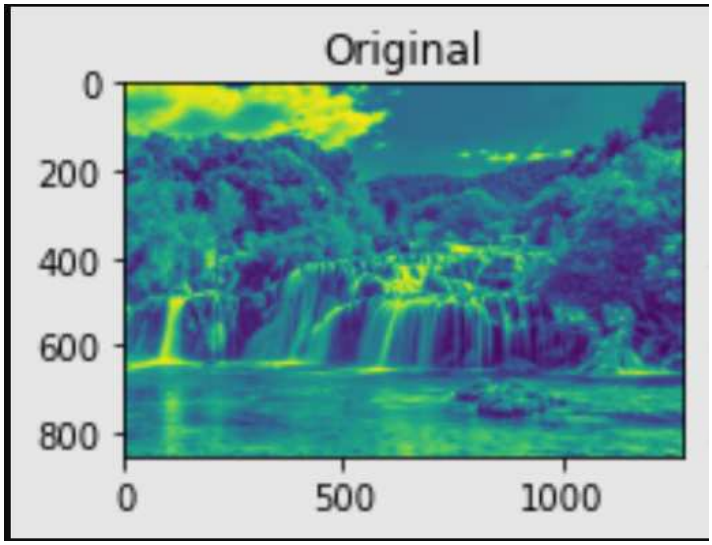
Slika 5.2: Ažuriranje varijabli za ispravak reziduala

```
def operator_smanjivanja(x, c, mu):
    return max(abs(x) - 2 * c / mu, 0) * np.sign(x)
```

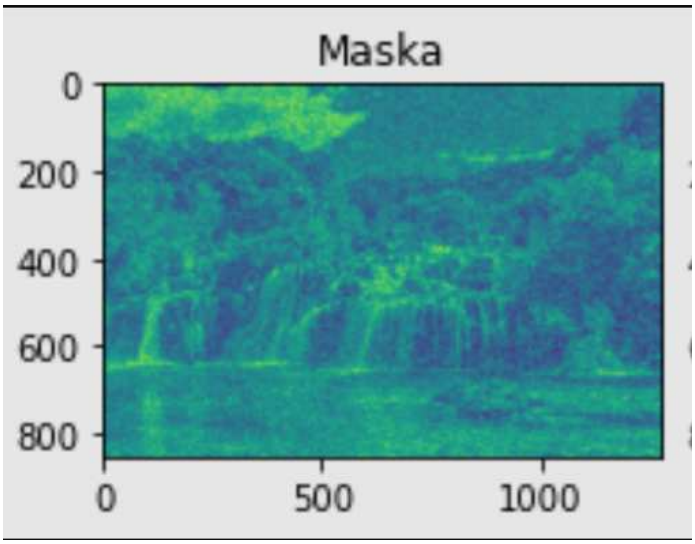
Slika 5.3: Korišteni operator smanjivanja

Rezultati

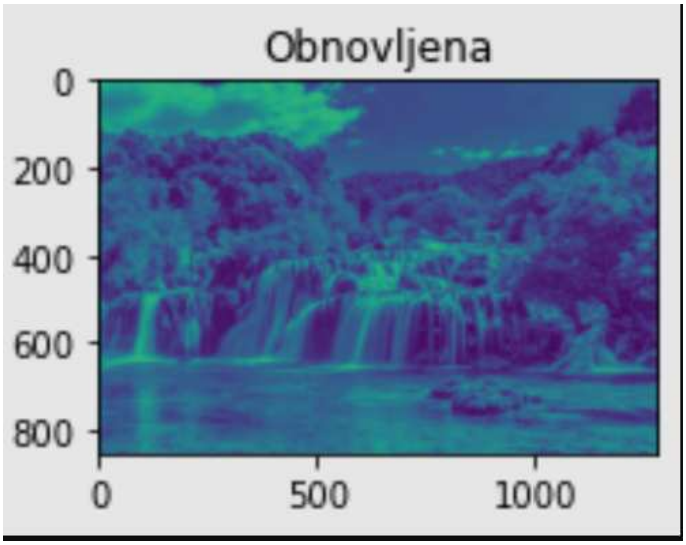
Sada ćemo vidjeti razliku u restauraciji nakon 50 i 100 iteracija na slici pokvarenoj Gaussovim šumovima na 50 posto piksela.



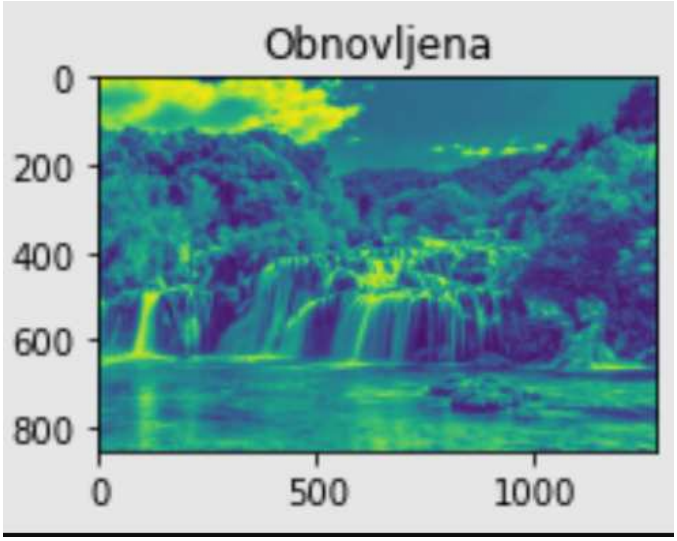
Slika 5.4: Originalna slika



Slika 5.5: Oštećena slika



Slika 5.6: Algoritam nakon 50 iteracija



Slika 5.7: Algoritam nakon 100 iteracija

Složenost algoritma

Matrica U je veličine $m \times r$, dok je matrica V $r \times n$. Složenost algoritma se nalazi većinski u ažuriranju te dvije varijable, naime, na njima se vrši prethodno spomenuti algoritam SVD pa je složenost računanja U $O(mr^2)$ te analogno za V $O(nr^2)$. Sumiranjem tih operacija dobijemo složenost algoritma HRMC (eng. Huber-based Robust Matrix Completion algorithm) od $O((m+n)r^2)$. U praktičnoj primjeni vrijedi $r << n$.

Poglavlje 6

Zaključak

Optimizacijski algoritmi, kao što je Adam, igraju ključnu ulogu u treniranju modela strojnog učenja. Ovi algoritmi iterativno ažuriraju parametre modela kako bi minimizirali funkciju gubitka i mogu imati značajan utjecaj na brzinu i točnost procesa treninga. Adam kombinira prednosti adaptivnih stopa učenja (poput Adagrada) i optimizacije temeljene na momentu. Dizajniran je za brzu i robusnu konvergenciju, čak i kada se radi o rijetkim gradijentima, što može biti uobičajeno u skupovima podataka u stvarnom svijetu. Prati prethodne gradijente i prilagođava brzinu učenja na temelju veličine i smjera gradijenta. Adaptivno prilagođava brzinu učenja svakog parametra. Relativno je jednostavan za korištenje, te postoje mnoge implementacije dostupne u popularnim bibliotekama strojnog učenja kao što su PyTorch, Autograd ili Jax. Općenito, algoritmi za optimizaciju su bitan alat kod strojnog učenja radi postizanja boljih performansi i brže konvergencije pri treniranju modela. Odabirom odgovarajućeg algoritma optimizatora i ugađanjem hiperparametara metode moguće je poboljšati točnost i učinkovitost te su primjenjivi na različitim zadacima strojnog učenja.

Bibliografija

[1] D. Bakić, *Linearna algebra*, 2008.

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne i Qiao Zhang, *JAX: composable transformations of Python+NumPy programs*, 2018, http://github.com/google/jax.

[3] I. Gogić, P. Pandžić i J. Tambača, *Diferencijalni racun funkcija vise varijabli*, 2001.

[4] A. Hayes, *Multicollinearity*, https://www.investopedia.com/terms/m/multicollinearity.asp.

[5] IBM, *Machine learning*, https://www.ibm.com/topics/machine-learning.

[6] Anastasiya Ivanova, Pavel Dvurechensky, Evgeniya Vorontsova, Dmitry Pasechnyuk, Alexander Gasnikov, Darina Dvinskikh i Alexander Tyurin, *Oracle Complexity Separation in Convex Optimization*, Journal of Optimization Theory and Applications **193** (2022), br. 1, 462–490, ISSN 1573-2878, https://doi.org/10.1007/s10957-022-02038-7.

[7] R. Khandelwal, *L1-L2 regularization*, https://medium.datadriveninvestor.com/l1-l2-regularization-7f1b4fe948f2.

[8] Diederik P Kingma i Jimmy Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980 (2014).

[9] Patrick Loeber, *Python Engineer*, https://www.python-engineer.com/courses/pytorchbeginner/05-gradient-descent/.

[10] Schneider R. i Mikulik V., *Jax-basics*, https://jax.readthedocs.io/en/latest/index.html.

[11] Lutz Roeder, *Netron*, https://pypi.org/project/netron/.

[12] Sebastian Ruder, *An overview of gradient descent optimization algorithms*, arXiv preprint arXiv:1609.04747 (2016).

[13] N. Sandrić i Z. Vondraček, *Vjerojatnost*, 2019.

[14] Mark Schmidt, *Mark Schmidt*, https://www.cs.ubc.ca/~schmidtm/.

[15] P. Sharpe, *Autograd*, https://github.com/HIPS/autograd/blob/master/docs/tutorial.md.

[16] L. Tang i W. Guan, *Robust matrix completion with complex noise*, (2019).

[17] Wikipedia, *Linear regression*, https://https://en.wikipedia.org/wiki/Linear_regression.

[18] _____, *Linear separability*, https://en.wikipedia.org/wiki/Linear_separability.

Sažetak

U ovom radu obrađene su stohastičke gradijentne metode. Analizu smo većinom sproveli na metodi linearne regresije. Svi opimizatori su zbog te svrhe implementirani. Iz primjera smo vidjeli da se u preciznosti i efikasnosti najviše iskazala metoda adaptivnog reduciranja varijance Adam. Potom, opisali smo biblioteke automatskog deriviranja koda Jax i Autograd te na osnovu njih uvidjeli znatnu olakšicu u primjenama optimizatora. Naposljetku, implementirali smo algoritam za restauraciju slika naziva HRMC.

Summary

This thesis deals with stochastic gradient methods. We mostly conducted the analysis using the linear regression method. All optimizers are implemented for this purpose. From the examples, we saw that the method of adaptive variance reduction Adam was the most accurate and efficient. Then, we described the automatic code derivation libraries Jax and Autograd, and based on them, we saw a significant improvement in optimizer applications. Finally, we implemented an image restoration algorithm called HRMC.

Životopis

Rođen sam 6. rujna 1992. godine u Novoj Gradiški. Školovanje započinjem u Osnovnoj školi Ljudevita Gaja u mjestu rođenja, gdje upisujem i opću gimnaziju. Istu sam maturirao 2011. godine. Prirodoslovno-matematički fakultet u Zagrebu upisujem 2014. godine na kojem započinjem Prediplomski studij Matematika koji završavam 2020., stekavši titulu baccalaureus matematike. Te godine upisujem na dotičnom fakultetu Diplomski studij Računarstvo i matematika.