Kratek opis strukture nevronske mreže - Razred Layer

Implementirala sem dva tipa slojev: *LinearLayer* in *ActivationLayer*. Oba tipa imata implementirano funkcijo *forward_propagation()* in *backward_propagation()*.

LinearLayer Da nebi prišlo do napake pri regularizaciji, hranim uteži in bias v *LinearLayer-ju* ločeno. Za lažji implementacijo *backward_propagation()* hranim tudi zadnji vhod.

Forward_propagation() Izhod za vsak nevron lahko enostavno izračunamo tako $y_j = b_j + \sum_i x_i w_{ij}$. Matrično to naredimo tako:

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} \dots w_{1j} & & \\ \vdots & \ddots & \vdots \\ w_{i1} \dots & w_{ij} & \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$

$$Y = XW + B$$

Backward_propagation() Predpostavimo lahko, da imamo matriko, ki vsebuje odvod napake glede na izhod trenutnega sloja $\frac{\partial E}{\partial Y}$. Potrebujemo torej odvod napake glede na parametre $\frac{\partial E}{\partial W}$, $\frac{\partial E}{\partial B}$ in odvod napake glede na vhod $\frac{\partial E}{\partial X}$. Po premisleku ugotovimo da lahko to izračunamo tako:

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^t$$

$$\frac{\partial E}{\partial W} = X^t \frac{\partial E}{\partial Y}$$

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}$$

Prav zaradi druge enačbe je bilo potrebno hraniti tudi zadnji vhod, ki ga je dobila funkcija *forward_propagation()*

ActivationLayer Implementacija *ActivationLayer* je precej enostavnejša. Hraniti moramo le nelinearno funkcijo (npr. Relu, tanh, softmax,...) in njen odvod.

Forward_propagation() Za dani vhod X je izhod preprosto aktivacijska funkcija, uporabljena za vsak element X. Kar pomeni, da imata vhod in izhod enake dimenzije.

$$Y = \begin{bmatrix} f(x_1) & \dots & f(x_i) \end{bmatrix} = f(X)$$

Backward_propagation() Glede na $\frac{\partial E}{\partial Y}$ želimo izračunati $\frac{\partial E}{\partial X}$. To matrično naredimo tako:

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} \odot f'(X)$$

Kratek opis strukture nevronske mreže - Razred Model V razredu model moramo inicializirati *loss funkcijo*, odvod *loss funkcije* ter seznam objektov *LinearLayer* in *ActivationLayer*. Klasifikacijski in regresijski model se razlikujeta zgolj v začetni inicializaciji.

Forward() Funkcija *forward()* sprejme en sam sample matrike X ter se sprehodi čez vse layer-je in kliče funkcijo *Forward_propagation()* za posamezni layer.

Backward() Funkcija *bacward()* sprejme en sam sample matrike y ter izračuna odvod loss funkcije se sprehodi čez vse layer-je in kliče funkcijo *Backward_propagation()* za posamezni layer.

Predict() Za potrebe testnih primerov sem implementirala še funcijo *predict()*, ki deluje enako kot *Forward()* le da sprejema matriko in vrača urejeno matriko.

Weights() Funkcija je prav tako implementirana za potrebe testnih primerov. Sprehodi se čez vse LinearneLayer-je ter prebere in shrani uteži W in biase B

Inicializacija in razlike pri klasifikacijskem in regresijskem modelu

Vse kar sem do sedaj opisala razen inicializacije si klasifikacijski in regresijski model delita. Sedaj bom natančneje opisala inicializacijo.

Loss funkcija Loss funkciji in njena odvoda sta seveda različna. Pri regresiji sem uporabila kar **MSE**, medtem ko pri klasifikaciji **logloss**.

Seznam objektov *LinearLayer* in *ActivationLayer* Inicializacijska funkcija sprejme seznam dimenzij linearnih layer-jev ki sem jih uporabila za inicializacijo seznama objektov layer-jev. Za vsako dimenzijo v seznamu sem definirala *LinearLayer*,ki raztegne ali pa skrči X glede na zahteve. Paziti sem morala da je začetna dimenzija število atributov, končna pa pri klasifikaciji število razredov ki jih morem napovedati, pri regresiji pa 1. Vsakemu *LinearLayer-ju* razen zadnjemu sledi *ActivationLayer*, z aktivacijsko funkcijo *tanh()*. Pri klasifikaciji sem na koncu dodala še *ActivationLayer* ki vzame za aktivacijsko funkcijo softmax.

Kompatibilnost gradient-a in cost-a

MSE Kompatibilnost sem utemeljila z numeričnim gradientom, ki sem ga izračunala z sledečo funkcijo. Generirala sem dva naključna vektorja a in b na katerih sem uporabila *mse_prime()* in *mse_prime_numerical()*. To je spodnja funkcija.

```
def mse_prime_numerical (a,b):
        eps = 1e-6
        d_num = []
        for i in range (10):
                b_add = copy.deepcopy(b)
                b_add[i]= b_add[i] + eps
                b_sub = copy.deepcopy(b)
```

```
                b_sub[i] = b_sub[i] - eps
                d_num.append((mse(a,b_add)-mse(a,b_sub))/(2*eps))
        return np.array(d_num)
```

Primerjala sem vektorja ki ju dobim z numeričnim in in nenumeričnim gradientom. Vektorja sta enaka.

```
        print(mse_prime(a,b).flatten())
        print(mse_prime_numerical(a,b))
        if (mse_prime_numerical(a,b).all() == mse_prime(a,b).flatten().all()):
        print("Je enako!")
        ...
        [-0.17832511  0.13620088  0.03113076  0.01980606 -0.13735116 -0.06443925
        -0.111171    0.055364    0.02484676  0.05482814]
        [-0.17832511  0.13620088  0.03113076  0.01980606 -0.13735116 -0.06443925
        -0.111171    0.055364    0.02484676  0.05482814]
        Je enako!
```

logloss Podobno sem naredila tudi za logloss. Funkcija je malo drugačna.

```
def log_los_prime_numerical(b,a):
        eps = 1e-5
        d = []
        for i in range (10):
                d1 = []
                for j in range(2):
                        a_add = copy.deepcopy(a)
                        a_sub = copy.deepcopy(a)
                        a_add[i,j] = a_add[i,j]+eps
                        a_sub[i,j] = a_sub[i,j]-eps
                        d1.append((log_loss(b, a_add)-log_loss(b,a_sub))/(2*eps))
                d.append(d1)
        return np.array(d)
```

Matriki sta ponovno enaki.

```
if (log_loss_prime(b,a).all() == log_los_prime_numerical(b,a).all()):
        print("Je enako!")
...
[[ 0.          -0.19897246]
[-0.2255087   0.         ]
[ 0.          -0.14323607]
        ...
[ 0.          -0.32154214]
[-0.22529651  0.         ]]

[[ 0.          -0.19897246]
[-0.2255087   0.         ]
[ 0.          -0.14323607]
        ...
[ 0.          -0.32154214]
[-0.22529651  0.         ]]
Je enako!
```

Primerjava z linearno regresijo in multinomsko logistično regresijo na podatkih *housing.csv*

Regresijska nevronska mreža vs. linearna regresija Mojo regresijsko nevronsko mrežo lahko konfiguriram tako, da ima samo eno linearni sloj. To je pravzaprav linearna regresija. Podatke sem razdelila na train in test. Model sem testirala za 4 različne konfiguracije nevronske mreže in logistično regresijo. Ostale konfiguracije so: $lambda = 0.0001$, $learning_rate = 0.0001$.

konfiguracija	MSE
linearna regresija:[]	29.75435
[20]	31.80033
[20, 10]	30.34702
[50, 20, 10]	39.41045
[70, 50, 20, 10]	46.54452

Zelo očitno je da pregloboke nevronske mreže močno prekomerno prilagodi trening množici. Vklopila sem *early_stoping = True*. S tem podatke med treniranjem razdelim na train in validation množico in shranim model, ki je najboljši na validation množici. Prav tako neham trenirat če se loss na trening množici spremeni za manj kot $\epsilon = 10^{-5}$.

konfiguracija	MSE
linearna regresija:[]	31.45791
[20]	30.23857
[20, 10]	29.34610
[50, 20, 10]	33.30603
[70, 50, 20, 10]	29.43142

Zaradi premajhnega števila primerov in preveč kompleksnih arhitektur nevronskih mrež tudi to ne pomaga prav veliko.

Klasifikacijska nevronska mreža vs. multinomska logistična regresija Podobno kot lahko konfiguriram tudi Klasifikacijsko nevronsko mrežo tako da ima zgolj en linearni sloj in softmax aktivacijo. Podatke sem razdelila na train in test. Model sem testirala za 4 različne konfiguracije nevronske mreže in logistično regresijo. Ostale konfiguracije so: $lambda = 0.0001$, $learning_rate = 0.0001$.

konfiguracija	**log loss**
multinomska logistična regresija:[]	0.36153
[20]	0.35041
[20, 10]	0.32377
[50, 20, 10]	0.31257
[70, 50, 20, 10]	0.28632

Če vključim *early_stoping = True* rezultati niso znatno boljši.

konfiguracija	**log loss**
multinomska logistična regresija:[]	0.34799
[20]	0.31475
[20, 10]	0.33381
[50, 20, 10]	0.29148
[70, 50, 20, 10]	0.29126

Gradnja modela in funkcija *fit()*

Funkcija *fit()* Funkcija, ki natrenira moj model sprejme naslednje argumente: epoch, learning_rate, verbose, early_stopping. Prebere pa lambdo in seznam dimenzij linearnih layerjev units. Seznam dimenzij najprej primerno preuredim gleda na to ali imam klasifikacijo ali regresijo, potem definiram model. Model treniram tolikokrat kot mi to veli spremenjlivka epoch. Če je spremenljivka early_stopping nastavljena na true potem predčasno X razdelim na train in validation. Posebej shranjujem modelj ki najbolje napoveduje na validation. Ta model tudi vračam. Prav tako se ustavim če se napaka na trening množici spremeni zelo malo. Lambdo in learning rate potrebuje funkcija backward. Verbose pa kot že pove ime izpisuje vmesne rezultate.

Gradnja modela Naredila sem grid search in ga testirala na validation znotraj fit. Preiskusila sem parametre $lambdas = [0.01, 0.001, 0.0001]$, $learning\_rates = [0.01, 0.001, 0.0001]$ in $layer\_configs = [[150, 100, 50, 20], [100, 50, 20], [50, 20], [20]]$. $Epoch = 100$ a sem nastavila early_stoping na True. Najboljših pet rezultatov in konfiguracije so:

konfiguracija	lambda	learning_rate	**log loss**	duration
[50, 20]	0.0001	0.01	0.54864	0 : 15 : 31
[100, 50, 20]	0.0001	0.01	0.55620	0 : 59 : 07
[20]	0.0001	0.01	0.57125	0 : 12 : 10
[150, 100, 50, 20]	0.0001	0.01	0.585635	1 : 22 : 41
[100, 50, 20]	0.0001	0.001	0.58800	1 : 04 : 59

Kot je razvidno iz tabele sta najboljša $lambda = 0.0001$ in $learning\_rate = 0.01$. Globina mreže na rezultat ne vpliva z jasnim vzorcem.

Za končen model sem torej uporabila $lambda = 0.0001$ in $learning\_rate = 0.01$ in trenirala 100 epochov. Mreža se je trenirala 1 : 03 : 29. Score na validation setu se ni izboljševal od 30 epocha naprej.