Scurta introducere

Rețelele neurale artificiale reprezinta niste ansambluri de elemente de procesare simple, puternic interconectate care opereaza în paralel si urmăresc să inregistreze informatia într-un mod asemănător creierului uman, fiind compuse din neuroni artificiali și prezentand capacitatea de a învăța. Acest lucru se realizeaza prin intermediul unor interconexiuni prin care se propagă informatia numerica.

Un CNN este o retea neurala compusa din mai multe straturi de convoluție. Spre deosebire de straturile din rețelele neuronale clasice, în straturile de convoluție numărul de conexiuni se reduce la dimensiunea filtrelor utilizate (in locul dimensiunii imaginii).

SVM-urile (rețelele vector-suport) sunt modele de învățare supravegheate cu

algoritmi de învățare asociați care analizează datele pentru clasificare și analiza de regresie.

Partea1 - CNN

Cele doua submisii pe care le-am selectat pentru concurs sunt prezente in MyModel.py . Am folosit acelasi model pentru amandoua doar ca pe prima am antrenat-o doar pe train data si am adaptat abordarea in functie de val\_loss - ul de pe validation data la fiecare pas (la fiecare epoca). Vezi run().

Pentru a doua submisie (cea finala cu scorul mai mare) am concatenat train data cu validation data si am antrenat modelul pe intregul set de date. Vezi run\_concatenated().

Nu am folosit grid search deoarece aproape mi-a omorat calculatorul.

In schimb am plecat de la un template de cnn si am adjustat/adaugat/eliminat valori in functie de comportamentul modelulu:

~ Am crescut *numarul de neuroni* la 64, 128, 128, 256 ca sa creasca accuracy-ul.

~ Am calibrat *bias\_regularizer* si *kernel\_regularizer*, am adaugat *constraints* si *dropout-uri* ca sa nu mai faca overfitting.

~ Am ales sa folosesc *functia de activare ELU* deoarece am citit intr-un research ca e venita ca o imbunatatire pentru Relu si in practica se comporta mai bine Elu fara normalizare decat Relu cu normalizare. Totusi am vazut ca daca pastrez un Relu urmat de BatchNormalization la una din convolutii imi da rezultate mai bune decat daca il elimin complet(iar daca il foloseam peste tot se purta mai prost).

~ Am adaugat un *learning rate adapter*:

rlrop = ReduceLROnPlateau(monitor=monitor, mode='auto', patience=2, verbose=1, factor=0.2)

deoarece pe la jumatate modelul incepea sa stagneze si valorile lui val\_loss deveneau din ce in ce mai apropiate; daca il lasam prea mult incepea chiar sa o dea in partea cealalta si sa creasca (odata cu val\_acc). Val\_acc ul stagna si el, pana cand incepeau sa scada impreuna.

Parametrul *monitor* primeste fie valoarea:

* “val\_loss” daca exista validation data si ma ghidez dupa rezultatele pe acest set de date pe parcursul antrenarii.
* “loss” daca nu am validation data ca parametru. ( Situatia in care concatenez validation data la train data si antrenez pe tot setul ).

Am ales *mode=’auto’* deoarece nu mereu trebuia doar sa creasca sau doar sa scada factorul de invatare. Asa s-a modificat adecvat momentului si s-a descurcat cel mai bine.

Pe *patience* am decis sa il reduc la 2 deoarece am vazut ca daca as vrea sa il las mai ingaduitor valorile nu s-ar remedia de la sine.

Pentru *factor* o valoare mai mare ar fi insemnat o discrepanta prea mare(invatarea n-ar mai fi fost accurrate), iar o valoare mai mica nu ar fi dat roade.

~ Am ales numarul de *epoci* testand mai multe valori (intre 7 si 20). Pe un numar de epoci:

* < 15 se oprea la un accuracy mai mic.
* > 15 valorile nu mai evoluau, ba chiar accuracy-ul ajungea sa scada putin, incepea sa faca si overfitting din nou si nu reuseam sa-l mai recalibrez.

Tabel cu cele mai bune solutii gasite:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Convolutional  neurons | Dense  neurons | Activation functions | Training data | monitor | Epochs  number | Kaggle  Final Result |
| 64, 128 | 256 | elu peste tot | train\_data | fara learning rate adapter | 12 | 0.64307 |
| 64, 64, 128, 128 | 256 | elu peste tot | train\_data | val\_loss | 16 | 0.72512 |
| 64, 64, 128, 128 | 256 | elu peste tot | concat([train\_data, valid\_data]) | loss | 16 | 0.75384 |
| 64, 128, 128, 256 | 256, 512 | elu,elu,elu, relu,elu | train\_data | val\_loss | 15 | 0.74153 |
| 64, 128, 128, 256 | 256, 512 | elu,elu,elu, relu,elu | concat([train\_data, valid\_data]) | loss | 15 | 0.76307 |

Obs: parametrii nementionati in tabel au ramas constanti in toate variantele. Doar cand am adaugat Dense(512, ..) i-am pus 0.0001 in loc de 0.0005 in regularizers.

Prelucrarea datelor

Citesc o imagine in felul urmator (o convertesc intr-un array si discretizez intervalul):

def read\_image(path):  
 *"""  
 Retin imaginea sub forma de array.  
 O impart la 255 deoarece sunt 256 de pixeli (de la 0 la 255) si o sa le asociez numai valori intre 0 si 1.* ***:param*** *path:* ***:return****:  
 """* img\_mat = image.load\_img(path, target\_size=(50, 50, 1), grayscale=True)  
 img\_mat = image.img\_to\_array(img\_mat)  
 img\_mat = img\_mat / 255  
 return img\_mat

Mai departe, adaug imaginile intr-o lista si convertesc lista tot la un np.array.

Partea2-SVM

Inainte de aceasta abordare cu cnn, am incercat mai multe SVM – uri, 4 dintre ele fiind prezente in SVM.py. Pentru acest fisier am si un fisier cu functii ajutatoare, anume main\_utils.py pentru citire, afisare, normalizare si adjustarea pozelor.

Tabel cu rezultate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C | Kernel | gamma | norm\_type | Acc |
| 1.0 | ‘linear’ | ‘auto’ | l2 | 38, 9333333333 |
| 1.0 | ‘rbf’ | ‘auto’ | l2 | 43.53333333333333 |
| 1.0 | ‘rbf’ | 10 | l2 | 43.37777777777777 |
| 1.0 | ‘poly’ | 7 | min-max | 43.044444444444444 |
| 1.0 | ‘poly’ | 7 | l2 | 43.77777777777778 |
| 1.0 | ‘poly’ | 6 | standard | 40.666666666666664 |

Observatii:

1. Am incercat mai multe combinatii decat sunt prezente in tabel (inclusiv cu normalizare standard sau l1, sau cu alte valori pentru C) doar ca au dat un accurracy chiar mai mic asa ca nu le am mai luat in considerare. Am introdus in tabel doar topul primelor 6 in functie de accurracy. Valoarea 1.0 pentru C si parametrul l2 pentru normalizare s-au purtat cel mai bine.

2. Acestea sunt accuracy-urile rezultate pe calculatorul meu. Pe kaggle accuracy–urile ar fi fost mai mici deoarece in medie, dupa incarcarea pe kaggle solutia scadea cu aproximativ 3%.

Cel mai mult am luat 43.7777 % accurracy (40.786 % pe kaggle, 38.666 % in final) cu urmatorul model:

def SVM4(train\_images, train\_labels, valid\_images):  
 print("incepe normalizarea")  
 train\_images, valid\_images = main\_utils.normalize\_data(train\_images, valid\_images, 'l2')

print('Creez SVC ul')  
 clf = SVC(C=1.0, gamma=7, kernel='poly')  
  
 print("incepe trainingul")  
 clf.fit(train\_images, train\_labels)  
  
 print('incep predictiile')  
 valid\_preds = clf.predict(valid\_images)

Am mai incercat si un Linear Discriminant Analysis si un Quadratic Discriminant Analysis ( Le-am lasat si pe acestea prezente in fisierul cu SVM-uri, deoarece e mai mult un fisier de incercari), dar cel mai mult am obtinut 38.2 pe QDA la mine pe calculator.

Main Utils

Main\_utils.py este un fisier cu functii ajutatoare pentru: citire, afisare, normalizare si adjustarea pozelor. Main Utils este folosit doar de fisierul SVM.py!

De aceasta data am avut retinute imaginile intr un mod diferit fata de cazul CNN. Dupa ce imi creez np.array-ul cu imagini, ii schimb forma, apeland functia reshape.

nsamples – reprezinta numarul de imagini

nx\*ny\*nz – reprezinta dimensiunile inmultite ale unei imagini

def reshape(images):  
 nsamples, nx, ny, nz = images.shape  
 images = images.reshape((nsamples, nx \* ny \* nz))  
 return images

Acum pot sa o transmit ca parametru pentru normalizare.

Mi am definit functia de normalizare in felul urmator:

def normalize\_data(train\_data, test\_data, norm\_type=None):  
 if norm\_type is None:  
 return train\_data, test\_data  
  
 if norm\_type == 'standard':  
 scaler = preprocessing.StandardScaler()  
 scaler.fit(train\_data)  
 return scaler.transform(train\_data), scaler.transform(test\_data)  
  
 if norm\_type == 'min\_max':  
 return (preprocessing.minmax\_scale(train\_data, axis=-1),  
 preprocessing.minmax\_scale(test\_data, axis=-1))  
  
 if norm\_type == 'l1':  
 return (preprocessing.normalize(train\_data, norm='l1'),  
 preprocessing.normalize(test\_data, norm='l1'))  
  
 if norm\_type == 'l2':  
 return (preprocessing.normalize(train\_data, norm='l2'),  
 preprocessing.normalize(test\_data, norm='l2'))