Documentație proiect clasificare imagini

Machine Learning

De Gherasim Rareș, grupa 243

Scop: Realizarea unui algoritm de clasificare care să aibă o precizie cât mai bună pe datele de validare și de testare, după antrenarea acestuia pe datele de antrenare, primite în cadrul competiei de pe kaggle.

Link Comepeție: https://www.kaggle.com/c/ai-unibuc-24-22-2021

Variabile utilizate pentru stocarea imaginilor:

1. train_images ➔ reține imaginile de antrenare;
2. validation_images ➔ reține imaginile de validare;
3. test_images ➔ reține imaginile de testare;

   Toate de tip np.array pentru a avea acces la metodele din librăria numpy.

Variabile utilizate pentru stocarea label-urilor:

1. train_labels ➔ reține label-urile pentru imaginile de antrenare;
2. validation_labels ➔ reține label-urile pentru imaginile de validare;

Modele de clasificare utilizate utilizate:

I. SKLEARN:
   a. MNB (Multinomial Naive Bayes)
   b. KNN Classifier (K-Nearest Neighbors)
   c. SVM Classifier (Support Vector Machine)

II. Neural Networks:
   a. CNN (Convolutional Neural Network)

Preprocesări realizate pe date înainte de clasificare.

Pentru început am încercat să normalizez datele cu l1 sau cu l2 precum am făcut la laborator, dar după ce am văzut faptul că predicțiile scădeau pe imaginile normalizate față de cele care nu erau normalizate am renunțat la normalizare.

Un alt motiv pentru care am renunțat la normalizare este faptul că, la afișare, matricile imaginilor conțineau doar valori între 0 și 1, practic semnificând faptul că acestea sunt normalizate deja sau cel puțin nu mai au nevoie de normalizare.

Exemplu (train_images[0]):

```
[[0.57254905 0.5764706  0.5647059  ... 0.5137255  0.50980395 0.50980395]
 [0.54901963 0.5568628  0.5529412  ... 0.54901963 0.54509807 0.54509807]
 [0.54509807 0.54509807 0.5529412  ... 0.5254902  0.5254902  0.5176471 ]
 ...
 [0.11764706 0.1254902  0.12156863 ... 0.14117648 0.15294118 0.10588235]
 [0.12156863 0.14117648 0.13725491 ... 0.12941177 0.12156863 0.1254902 ]
 [0.14117648 0.13725491 0.14117648 ... 0.11372549 0.1254902  0.11372549]]
```

Preprocesările realizate diferă de la tipul de soluție abordată (clasificator predefinit în SKLEARN sau rețea neuronală):

I. SKLEARN

Singura preprocesare realizată este convertirea vectorilor de imagini din vectori reprezentați prin 3 dimensiuni (nr_imagini, width_img, height_img) în vectori de 2 dimensiuni, care vor fi de forma următoare: **(nr_imagini, width_img * height_img)**. Aceasta este necesară deoarece clasificatorii din SKLEARN nu primesc vector cu mai mult de 2 dimensiuni. Realizez asta prin apelarea unei functii predefinite pe fiecare vector, funcția fiind:

```python
def convertImagesTo2D(images):
    # Because the array is 3D, I need to reshape it into a 2D one in order for the classification to work.
    nr_of_images, x_axis, y_axis = images.shape
    images = images.reshape((nr_of_images, x_axis * y_axis))
    return images
```

II. Neural Network

În cazul rețelelor neuronale, numărul dimensiunilor trebuie să crească, acesta indicând și culorile pe care le poate avea imaginea (1 în cazul nostru pentru imagini alb-negru), pe lângă dimensionalitate lor 3D deja existentă: (nr_imagini, width_img, height_img).

```python
# I need to reshape the images because my keras layers require the input to have 4 dimensions.
# The input should have the following shape : (nr_of_images, length, height, color_chanels)
train_images = train_images.reshape(30001, 32, 32, 1)
validation_images = validation_images.reshape(5000, 32, 32, 1)
test_images = test_images.reshape(5000, 32, 32, 1)
```

MNB Classifier (Multinomial Naive Bayes)

Deoarece datele noastre (valorile pixelilor) sunt valori continue, va trebui sa le transformăm în valori discrete cu ajutorul unei histograme. Numărul de intervale la care vom împărți lungimea intervalului valorilor continue, apoi vom asigna fiecărei valori continue indicele intervalul corespunzător. Capetele intervalului le-am setat între 0 și 1 deoarece valorile pixelilor sunt între 0 și 1.

În cod am **încercat împărțirea datelor în i intervale**, pentru a observa cum variază acuratețea rezultată. Funcția put_values_in_bins avea rolul de a pune datele în nr_bins intervale.

Code:

```
def put_values_in_bins(matrix, nr_bins):
    ret = np.digitize(matrix, nr_bins)
    return ret


best_bin = 1
max_acc = 0.0
for i in range(1, 251):
    num_bins = i
    # Stop is equal to 1 because all pixels in the images have values in the interval
[0,1], so 1 is the maxim
    # value of a pixel and 0 is the minimum one
    bins = np.linspace(start=0, stop=1, num=num_bins)

    train_images_dig = put_values_in_bins(train_images, bins)
    validation_images_dig = put_values_in_bins(validation_images, bins)

    model_MNB = MultNB()
    model_MNB.fit(train_images_dig, train_labels)
    predictions = model_MNB.predict(validation_images_dig)
    print(i, " ", accuracy_score(validation_labels, predictions))
    if max_acc < accuracy_score(validation_labels, predictions):
        max_acc = accuracy_score(validation_labels, predictions)
        best_bin = num_bins

print("\n Maximum accuracy was recorded at ", best_bin ," number of bins. Accuracy of ",
max_acc, " .")
```

Rezultate: *(nr_of_bins – accuracy)*

bins: 1 – Accuaracy: 0.1108

bins: 2 -- Accuaracy: 0.1622

bins: 3 -- Accuaracy: 0.3604

bins: 4 -- Accuaracy: 0.3814

bins: 5 -- Accuaracy: 0.3876

bins: 6 -- Accuaracy: 0.3916

bins: 7 -- Accuaracy: 0.3924

bins: 8 -- Accuaracy: 0.3908

bins: 9 -- Accuaracy: 0.3922

bins: 10 -- Accuaracy: 0.3914

bins: 11 -- Accuaracy: 0.393

bins: 12 -- Accuaracy: 0.3936

bins: 13 -- Accuaracy: 0.3924

bins: 14 -- Accuaracy: 0.3916

Din rezultate putem observa ca acuratețea crește de la 1 interval până la 6, după care stagnează și rămâne constantă la ~0.39. Rezultatele ulterioare până la 250 de intervale respectau regula, nedepășind maximul de 0.3936 al acurateții.

Fără utilizarea intervalelor(bins) acuratețea clasificatorului este de 0.3898 pe datele de validare.

Confusion_Matrix (bins = 12):

```
[[ 61  86  26  46  88  24  43 103  93]
 [ 16 237  23  43  14  20  26 104  44]
 [ 20  65 147  25  47  95  33  80  21]
 [ 17  59   9 304  31  63  23  48  24]
 [ 19  50  40  38 180  71  21 107  28]
 [  4  15  31  56  38 334  13  54  16]
 [ 50  63  10  48  91  28 146  68  76]
 [ 20  54  19  32  52  21  19 288  15]
 [ 50  49  14  48  50  43  33  38 252]]
```

Classification_report:

	precision
0	0.24
1	0.35
2	0.46
3	0.47
4	0.30
5	0.48
6	0.41
7	0.32
8	0.44

Din matricea de confuzie și din raportul de clasificare putem sa vedem că nu este o clasa în particular pe care algoritmul să o prezică bine înafară de claseel 2,3,5, care au fost prezise aproape în 50% din cazuri, predicțiile fiind destul de slabe. Clasa 0 de exemplu fiind prezisă corect în 24% din cazuri.

KNN Classifier (K-Nearest neighbors)

Ideea clasificatorului: Pe scurt, clasificatorul pornește de la presupunerea că elementele de același fel se află aproape una de cealaltă, căutând n – cei mai apropiați vecini și clasificând obiectele noi pe baza clasificării celor mai apropriații n – cei mai aproapiați de acestea.

Am testat clasificatorul pe multimea de validare cu număr variabil de vecini (***n_neighbors***), **între 1 și 30**, iar din punct de vedere al rezultatelor am reținut doar cel mai bun rezultat și respectiv cel mai slab rezultat pentru fiecare ***p*** (tip de distanță) în parte.

Rezultate:

- Urmatoarele date sunt pentru p = 1 (manhattan_distance):
  - **Pentru 1 vecin avem rezultatul: 0.4492**
  - **Pentru 10 vecini avem rezultatul: 0.5064**
- Urmatoarele date sunt pentru p = 2 (euclidean_distance):
  - **Pentru 1 vecini avem rezultatul: 0.4416**
  - **Pentru 12 vecini avem rezultatul: 0.4908**

După ambele testări am observat faptul că acuratețea predicției pe setul de validare creștea de la 1 vecin treptat până când ajungeam la 10 vecini în cazul distanței manhattan sau 12 în cazul distanței euclidiene, după care începea să scadă din nou.

Code:

```
for j in range(1, 3):
        for i in range(1, 30):
                KNN = KNeighborsClassifier(n_neighbors=i, p=j)
                KNN.fit(train_images, train_labels)
                print(i, KNN.score(validation_images, validation_labels))
```

Confusion_Matrix (n_neighbors = 10, p = 1):

```
[[237  19   3  14  36   7  99  32 123]
 [ 51 253   9  17  19  10  64  49  55]
 [ 75  28 115  26  35  53  97  64  40]
 [ 52  18   8 335  22  33  49  13  48]
 [ 84  24  18  54 225  26  38  30  55]
 [ 30   7   8  62   9 336  44  26  39]
 [ 43   9   0  16   6  11 399  14  82]
 [ 64  27   5  34  32  14  74 227  43]
 [ 49   5   1   9   5   7  88   8 405]]
```

Classification_report:

	precision
0	0.35
1	0.65
2	0.69
3	0.59
4	0.58
5	0.68
6	0.42
7	0.49
8	0.46

Față de MNB putem observa îmbunătățiri destul de semnificative, precizia pe setul de validare fiind de peste 50% pe mai multe clase și chiar aproape 70% pe clasa 2. Între timp, încă există o deficiență în precizirea clasei 0, această fiind problematică și pentru MNB.

SVM Classifier

Am antrenat un clasificator de timp svm.SVC pe mulțimea de antrenarea. Cei doi parametrii foarte importanți în predicția datelor, tipul de kernel și parametrul de regularizare, am ales să îi testez pentru diferite valori. Pentru kernel am decis să parcurg toată lista de kernel din sklearn.svm.SVC, iar pentru parametrul de regularizare am parcurs toate valorile dintre 0.5 și 10, din 0.5 în 0.5. Rezultatele le-am împărțit pentru fiecare kernel în parte.

Rolul kernelul în cadrul acestui clasificator este de a transforma un model liniar, care de cele mai multe ori nu poate fi separat liniar, într-un model neliniar, în mai multe dimensiuni, pentru a-l putea separa liniar. Parametrul de regularizare C are rolul de a decide ce margine de eroare poate avea separarea liniară realizată de clasificator.

Code:

```
kernels = ['linear', 'rbf', 'poly', 'sigmoid']
cAux = [0.01, 0.5, 0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5,
7, 7.5, 8, 8.5, 9, 9.5, 10]

for eachKernel in kernels:
    print("\n++++++++++++++++++++++++++++++++++++++++++++++++++")
    print("++++++++++++++KERNEL ", eachKernel, "++++++++++++++++++++")
    for eachC in cAux:
        SVM_model = svm.SVC(C=eachC, kernel=eachKernel)
        SVM_model.fit(train_images, train_labels)
        predictions = SVM_model.predict(validation_images)
        print("C=", eachC, " Accuracy:", accuracy_score(validation_labels, predictions))

    print("++++++++++++++++++++++++++++++++++++++++++++++++++")
```

- **Kernel = „linear" – încearcă separarea prin linii drepte.**

Primul Kernel pe care l-am folosit pentru clasificare a fost cel liniar. În cazul acestuia am putut observa că acuratețea a crescut puțin de la ~0.001 până la ~0.01 (aproximez deoarece nu pot fi 100% sigur de parametrul cel mai optim), de unde a început să scadă treptat

C= 0.001 Accuracy: 0.5986

C= 0.005 Accuracy: 0.626

C= 0.009 Accuracy: 0.626

C= 0.01 Accuracy: 0.6366 (cea mai bună acuratețe obținută)

C=0.05 Accuracy: 0.6306

C= 0.1 Accuracy: 0.6218

C= 0.5 Accuracy: 0.6078

C= 1.0 Accuracy: 0.6016

C= 1.5 Accuracy: 0.596

(.....)

C= 7 Accuracy: 0.5702 (după aceasta valoare am oprit testarea deoarece dura foarte mult procesarea, iar trendul era clar descrescător)

- **Kernel = „sigmoid"**

Cel de al doilea Kernel pe care l-am folosit a fost cel sigmoid. Acuratețea acestuia a fost extrem de scăzută pe tot setul de date, din acest motiv nu am încercat să lucrez prea mult asupra lui și doar am atașat câteva exemple din rezultatele obținute. Variațiile au fost foarte mici, pentru C între 0.01 și 7.5, iar accuratețea foarte slabă.

C= 0.01 Accuracy: 0.1372

C= 0.5 Accuracy: 0.124

C= 0.1 Accuracy: 0.1264

C= 0.5 Accuracy: 0.124

C= 1.0 Accuracy: 0.1232

C= 1.5 Accuracy: 0.1234

(.....)

C= 6.5 Accuracy: 0.1226

C= 7 Accuracy: 0.1226

C= 7.5 Accuracy: 0.1224

- **Kernel = „poly”**

Cel de al treilea Kernel folosit, cel polynomial, se bazează pe o funcție polinomiala de grad 3 în cazul meu (default este egal cu 3 și când am schimbat gradul a scăzut acuratețea). Din rezultatele obținute se poate observa că acuratețea este mult mai bună ca la kernelul linear sau cel sigmoid, acestea învârtinudu-se în jurul valorii de 70%-71%. După C=5, rezultatele oscilează în jurul valorii de 71%, cel puțin până la ultima valoarea a lui C testată 10.

C= 0.01 Accuracy: 0.604	**C= 4.0 Accuracy: 0.7154**
C= 0.5 Accuracy: 0.7072	**C= 4.5 Accuracy: 0.7134**
C= 0.1 Accuracy: 0.6844	**C= 5.0 Accuracy: 0.7124**
C= 0.5 Accuracy: 0.7072	**C= 5.5 Accuracy: 0.7114**
C= 1.0 Accuracy: 0.7134	**C= 6.0 Accuracy: 0.7098**

(.....)

- **Kernel = „rbf”**

Ultimul Kernel testat a fost cel „rbf”. Acesta merge pe un principiu asemănător cu KNN, calculând distanțe între puncte, doar că față de acesta este mai eficient din punct de vedere al complexității spațiale deoarece își stochează rezultatele într-un vector suport. De altfel, cu ajutorul acestui kernel am obținut cea mai bună acuratețe pe setul de date folosind clasificatorul SVM, respectiv acuratețea de **75.7%** pe setul de validare.

Următoarele date au fost obținute pe setul de validate, cu kernel=’rbf’ și C variabil. Din rezultate putem observa faptul că acuratețea crește puternic până în jurul valorii de C = 4.5, după care scade puțin și rămâne constantă în jurul valori de **75.5%**

C= 0.01 Accuracy: 0.485	C= 5.0 Accuracy: 0.7558
C= 0.5 Accuracy: 0.718	C= 5.5 Accuracy: 0.754
C= 0.1 Accuracy: 0.64	C= 6.0 Accuracy: 0.7542
C= 0.5 Accuracy: 0.718	C= 6.5 Accuracy: 0.7548
C= 1.0 Accuracy: 0.735	C= 6.5 Accuracy: 0.7548
C= 1.5 Accuracy: 0.7448	C= 7 Accuracy: 0.7542
C= 2.0 Accuracy: 0.7502	C= 7.5 Accuracy: 0.755
C= 2.5 Accuracy: 0.753	C= 8 Accuracy: 0.7554
C= 3.0 Accuracy: 0.7544	C= 8.5 Accuracy: 0.7546
C= 3.5 Accuracy: 0.7544	C= 9 Accuracy: 0.7558
C= 4.0 Accuracy: 0.7562	C= 9.5 Accuracy: 0.7546
C= 4.5 Accuracy: 0.757	C= 10 Accuracy: 0.7554

Confusion_Matrix (C = 4.5, kernel='rbf'):

```
[[365  18  18  12  47   4  33  39  34]
 [ 21 436  11  10   7   6   7  23   6]
 [ 14  28 392  16  30  27   4  18   4]
 [ 25  16  20 421  19  21  14  25  17]
 [ 33  20  24  26 402  10   3  25  11]
 [  7   6  17  25  15 463   5  21   2]
 [ 27  13   9  11   7   6 467   8  32]
 [ 40  13  19  27  26  13   8 366   8]
 [ 25   5   4   8   3   8  41   7 476]]
```

Din matricea de confuzie putem observa faptul că, deși încă niciuna dintre valorile din aceasta nu este egală cu 0, totuși erorile din punct de vedere al predicției sunt tot mai mici, având foarte multe dintre valorile din matrice situate sub 10 predicții greșite.

Classification_report (C = 4.5, kernel='rbf'):

	precision
0	0.66
1	0.79
2	0.76
3	0.76
4	0.72
5	0.83
6	0.80
7	0.69
8	0.81

Precum s-a putut observa și în matricea de confuzie, clasele sunt prezise mult mai bine decât la modele de până acum, majoritatea fiind prezise cu o acuratețe de peste 75% pe setul de validare. Totuși, precum și la celelalte modele testate până acum, acuratețea clasificatorului este mult mai slabă pe imaginile ce aparțin de clasa 0, dar și de clasa 7.

GridSearchCV cu svm.SVC

```
from sklearn import svm
from sklearn.model_selection import GridSearchCV

params = {'C': [0.01, 0.5, 0.1, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5,
6.0, 6.5, 7, 7.5, 8, 8.5, 9, 9.5, 10],
          'kernel': ['rbf', 'poly', 'sigmoid', 'precomputed']}

print("\n++++++++++++++++++++++++++++++++++++++++++++++++++")
print("++++++++++++++KERNEL ", 'rbf', "++++++++++++++++++++")
aux = svm.SVC()
SVM_model = GridSearchCV(aux, n_jobs = -1, param_grid = params, cv = 2, refit = True,
verbose = 5)
SVM_model.fit(train_images, train_labels)
predictions = SVM_model.predict(validation_images)
print(classification_report(validation_labels, predictions))
print("++++++++++++++++++++++++++++++++++++++++++++++++++")
```

GridSearchCV este o metodă prin care poți testa toate combinațiile posibile de parametrii pe un clasificator ales de tine. De exemplu, în cazul codului scris de mine și rulat mai sus, GridSearchCV are rolul de a preluat modelul trimis de mine prin parametrul *aux* și al testa cu parametrii transmiși prin *params.*

Pentru a-și testa acuratețea, modelul împarte setul de antrenare în numărul de cv-uri pe care i-l dau (în cazul meu 2, adică în jumate) , după care se antrenează pe una dintre jumătăți și se testează pe cealaltă jumătate. La final, reține acuratețea pe care a avut-o după test și trece la următoarea combinație posibilă de parametrii.

Rularea acestuia se finalizează după ce reușește să parcurgă toate combinațiile posibile dintre valorile din *param_grid*.

Din punct de vedere al timpului utilizat, după parerea mea, metoda este destul de ineficientă, dar poate produce rezultate foarte bune dacă timpul nu este o problemă.

Rezultatul cel mai bun, cu algoritmul de GridSearchCV prezentat mai sus a fost obținut cu parametrii :

- C = 4.5
- Kernel = „rbf”

Rezultatul era de așteptat, deoarece același rezultat l-am obținut si când am parcurs aceeași parametrii doar că utilizând două for-uri și nu GridSearchCV.

Classification_report (GridSearchCV pe datele de validare):

	precision
0	0.66
1	0.79
2	0.76
3	0.76
4	0.72
5	0.83
6	0.80
7	0.69
8	0.81

Neural Networks : Convolutional Neural Network

Code:

```
Epoches = 16
Batch_size = 128

model = models.Sequential([
    layers.Conv2D(32, (5, 5), activation='relu', input_shape=(32, 32, 1)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (5, 5), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (5, 5), activation='relu'),

    layers.Flatten(),

    layers.Dense(128, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(9, activation='softmax')
])

model.compile(optimizer='adam',
              loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Pentru realizarea modelului de mai sus am folosit libraria tensorflow, respectiv API Keras care se află în aceasta. Din Keras ma extras layer-ele din cadrul modelului, dar și tipul de model (Sequential).

Arhitectura acestui model de CNN este formată din următoarele:

- Un layer convolutional cu 32 de filtre de mărimea 5x5 și cu functie de activare „relu”;
- Un layer de pooling care alege mereu valoarea maximă, de mărimea 2x2;
- Un layer convolutional cu 64 de filtre de mărimea 5x5 și cu funcție de activare „relu”;
- Un layer de pooling care alege mereu valoarea maximă, de mărimea 2x2;
- Un layer convolutional cu 128 de filtre de mărimea 5x5 și cu funcție de activare „relu”;
- Un layer de tip Flatten care reduce multidimensionalitatea vectorului de caracteristici la un vector unidimensional;
- Un layer de tip Dense de 128 de unități;
- Un layer de Dropout cu 0.2 unități pentru evitarea overtraining-ului;
- Un ultim layer de tip Dense cu 9 unități (numărul de clase posibile) și cu funcție de activare „softmax”.

Layer-ele de tip Conv2D aplică operațiile de convoluție pe imaginile date, iar cele de Pooling au rolul de a selecta feature-rurile cele mai semnificative pentru modelul nostru.

Rezumatul modelului:

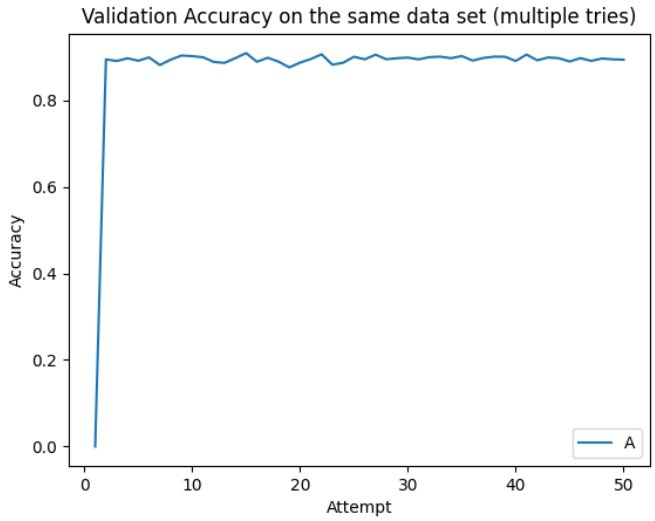
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 64)        51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 1, 1, 128)         204928
_________________________________________________________________
flatten (Flatten)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 128)               16512
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 9)                 1161
=================================================================
Total params: 274,697
Trainable params: 274,697
Non-trainable params: 0
_________________________________________________________________
```

Acest model mi-a oferit și eficiența cea mai bună pentru setul de date pe care eu am reușit să o obțin, respectiv de 91% pe setul de validare și 92% pe setul de testare.

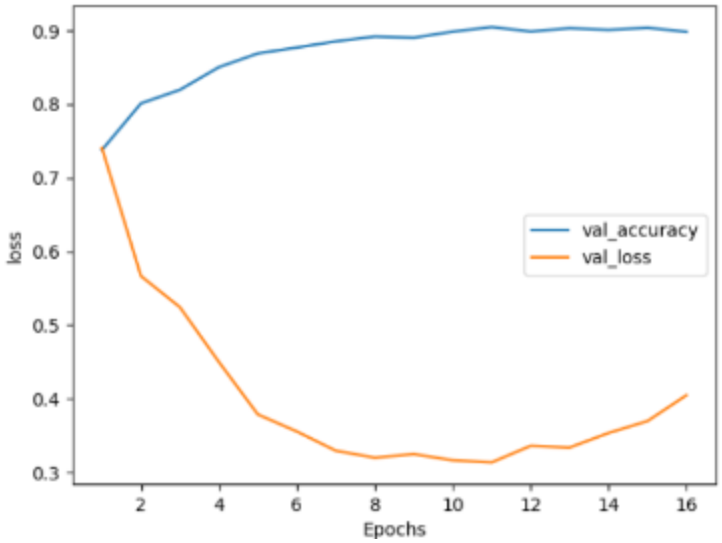
	precision	recall	f1-score	support
0	0.89	0.85	0.87	598
1	0.95	0.93	0.94	541
2	0.92	0.86	0.89	572
3	0.89	0.93	0.91	553
4	0.92	0.84	0.88	605
5	0.92	0.92	0.92	560
6	0.93	0.94	0.93	576
7	0.86	0.93	0.89	483
8	0.86	0.97	0.91	512
accuracy			0.91	5000
macro avg	0.91	0.91	0.91	5000
weighted avg	0.91	0.91	0.90	5000

Classification report pentru modelul prezentat pe setul de validare.

Un lucru foarte interesant pe care l-am observat la acest model este faptul că acuratețea sa variază de la o rulare la alta, putând să atingă minime de 89% acurateţe pe setul de validare și maxime de 91% pe același set.



Acuratețea modelului pe date de validare (linia albastră) nu este fixă de o rulare la altă. (50 de încercări)



Evoluția acurateței pe mulțimea de validare și a funcției de loss cu antrenarea epochilor.

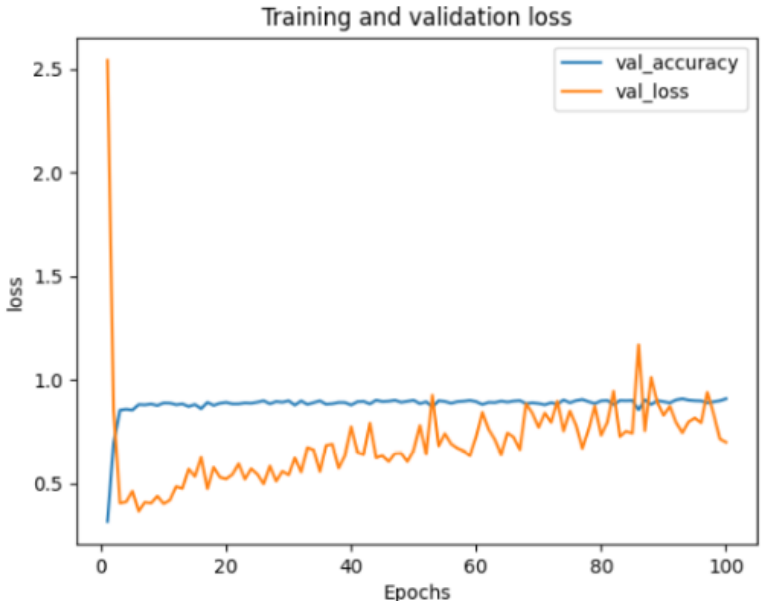
```
[[490   7   4   3  34   2   6  18   6]
 [  3 505   3   3   6   2   0   4   1]
 [  6  10 462   8  26  13   2   6   0]
 [  6   3   6 518  21   6   2  10   6]
 [ 11   2   9  12 502   3   2  10   3]
 [  4   1  10  13  16 497   4  16   0]
 [ 14   6   3   1  11   5 528   6   6]
 [  6   9   8  15   4   8   4 466   0]
 [ 15   3   0   5   8   2  14   5 525]]
```

Matrice de confuzie. Majoritatea valorilor care indică clasificare greșită sunt sub 10.

Alte încercări:

1. Am încercat antrenarea unui model cu mărimea filtrelor mai mici pe stratele de convoluție, respectiv de 3x3, deoarece am considerat că imaginile sunt deja destul de mici 32x32 încât ar trebui folosite filtre mai mici. După câteva rulări am constat că acuratețea pe mulțimea de validare a scăzut la 87% din cauza filtrelor mai mici, motiv pentru care am revenit la filtre de 5x5.

2. Am încercat utilizarea unui layer de normalizare de tip Batch (**BatchNormalization**) după fiecare MaxPool din model, dar și utilizarea a **100 de epochi**. Deși acuratețea rezultată avea valoare de 91% pe setul de validare, rezultatul pe setul de test a fost slab, respectiv 89%, ceea ce m-a făcut să cred că modelul a făcut overtraining pe datele de antrenare și cumva acelea semănau cu cele de validare, motiv pentru care performanța diferă așa de drastic.



Se poate observa că funcția de loss crește puternic, chiar dacă validare rămâne constantă (pe setul de validare), fapt ce indică prezența overtraining-ului.