

Brain Anomaly Detection

Mocanu Radu

Group 244

My name is Mocanu Radu, I am a 2nd year student at group 244, and in the following, I will document the approaches used to train deep learning models to sort the images from the test set of the "Brain Anomaly Detection" competition , from the Kaggle platform.

Some universal methods:

In order to process the data from the proposed archive and display certain metrics, I defined some methods that I used, in some places with small variations, in all the trials approached.

add_zeros->receives as a parameter the number of the image to be read and returns (in the form of a string) its correct name.

```
# adauga zerouri imaginii astfel incat sa se potriveasca cu formatul provided
def add_zeros(image_number):
    zeros_to_add = 6 - len(image_number)
    zeros = zeros_to_add * "0"
    return zeros + image_number
```

get_labels ->receives as parameters the number of labels to be read and the path where the unzipped file is located. The method returns a list of these.

```
# citeste labelurile din fisier
def get_labels(number_of_rows, path):
    train_labels = []
    fisier = open(path, 'r')
    fisier.readline()
    for i in range(number_of_rows):
        img_number, label = [int(i) for i in fisier.readline().split(",")]
        train_labels.append(label)
    fisier.close()
    return train_labels
```

get_images -> receives as parameters: the indexes of the images to be read (ie for the training images it will receive 0, 15000), the path where the file containing the images is found, the number of images to be read at a given time (so as not to work with very large lists, as the operations become slower and slower). The method reads the images and also converts the color images into black and white, since there are no relevant features in the RGB format.

```
# citește imaginile din fisierul dezarhivat
def get_images(start, number_of_images, base_path, number_of_images_to_process_at_a_time):
    processed_images = []
    total_images = []
    for i in range(start, start + number_of_images):
        image_name = add_zeros(str(i + 1))
        # convertește imaginea din rgb în alb negru
        img = Image.open(base_path + image_name + '.png').convert('L')
        # redimensionez imaginea 2D
        img_arr = np.array(img).reshape(-1)
        processed_images.append(img_arr)
        if i % number_of_images_to_process_at_a_time == 0:
            print(i)
            # întorc doar un număr prestabilit de imagini la un moment dat pentru a nu da exceed memoria RAM
            total_images = total_images + processed_images
            # golesc lista
            processed_images = []
    # verific dacă am ajuns la finalul imaginilor de citit
    if len(processed_images):
        return np.array(total_images + processed_images)
    return np.array(total_images)
```

createCsvFile -> receives as parameters the name of the file to be created and the list of predictions. The method creates the prediction file in csv format.

```
# creează fisierul de predicții în format csv
def createCsvFile(csvFileName, predictions):
    with open(csvFileName, "w") as f:
        print("id,class", file=f)
        predict_index = -1
        for i in range(17001, 22150):
            predict_index += 1
            print("0" + str(int(i)) + "," + str(int(predictions[predict_index])), file=f)
        f.close()
```

getConfusionMatrix -> receives as parameters the predictions of the model and the correct labels and returns the confusion matrix (using `confusion_matrix` from `sklearn.metrics`)

```
37] # intoarce matricea de confuzie
def getConfusionMatrix(validation_labels, classes_predictions):
    conf_matrix = confusion_matrix(validation_labels, classes_predictions)
    return conf_matrix
```

getRecallValue -> receives as parameters the confusion matrix and the number of the class for which we want to calculate the recall. The method returns the respective recall.

```
# intoarce valoarea recall-ului pe clasa specificata
def getRecallValue(conf_matrix, classNumber):
    if classNumber == 0:
        return conf_matrix[0][0] / (conf_matrix[0][0] + conf_matrix[0][1])
    return conf_matrix[1][1] / (conf_matrix[1][1] + conf_matrix[1][0])
```

getPrecisionValue -> receives as parameters the confusion matrix and the number of the class for which we want to calculate the precision. The method returns the respective precision.

```
# intoarce valoarea preciziei pe clasa specificata
def getPrecisionValue(conf_matrix, classNumber):
    if classNumber == 0:
        return conf_matrix[0][0] / (conf_matrix[0][0] + conf_matrix[1][0])
    return conf_matrix[1][1] / (conf_matrix[1][1] + conf_matrix[0][1])
```

getAccuracyValue -> receives as a parameter the confusion matrix. The method returns the accuracy of the model that predicted that matrix.

```
# intoarce valoarea acuratetei pe clasa specificata
def getAccuracyValue(conf_matrix):
    return (conf_matrix[0][0] + conf_matrix[1][1]) / np.sum(conf_matrix)
```

Useful technique:

Due to a weak internet connection, I initially encountered great difficulties in trying to add the proposed archive to the collaborative working environment. Since the process was difficult and took a long time, every time the Internet connection was interrupted, I had to restart the entire process of uploading the archive from the local. Thus, I discovered a useful workaround, which I used every time. I uploaded the archive to my personal google drive account, google collab having the possibility of integrating the 2 platforms. Thus, loading the archive into the workspace was no longer a problem, co-unarchiving the images directly from my drive.

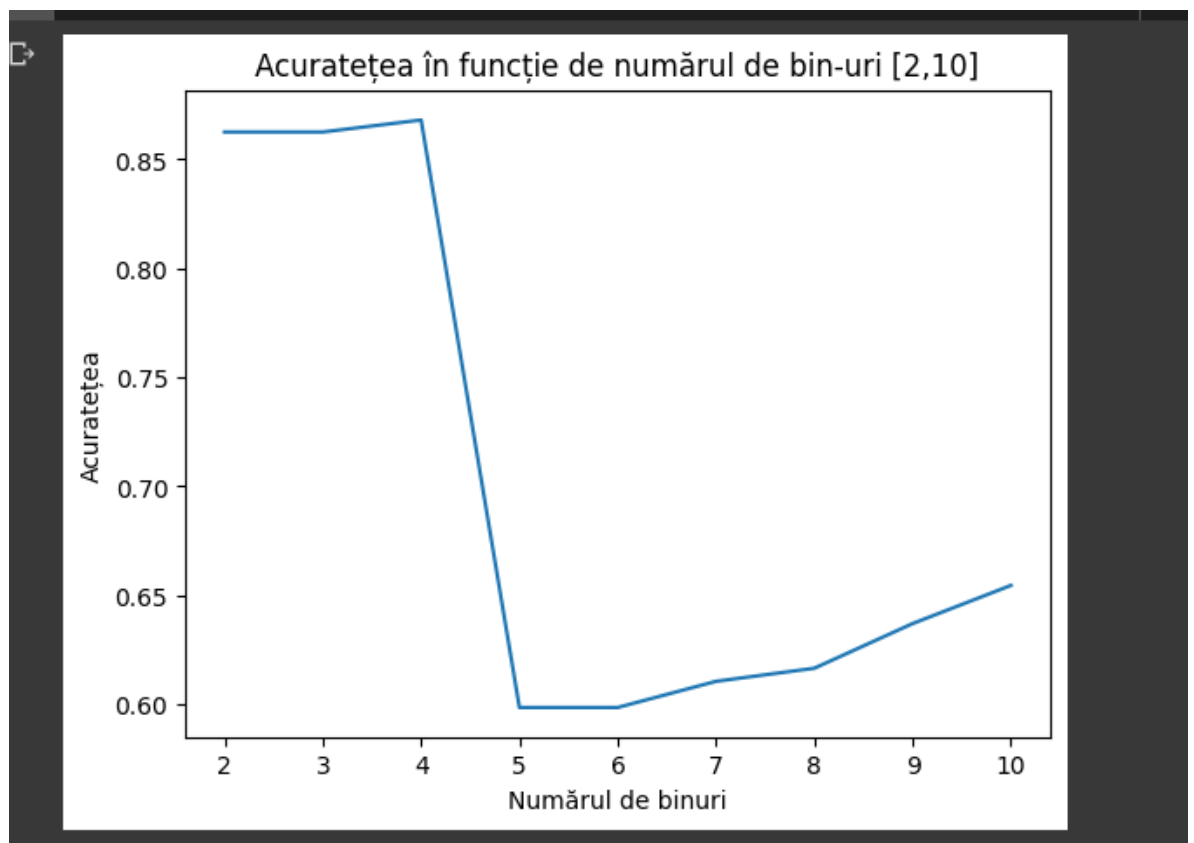
```
▶ from google.colab import drive  
drive.mount('/content/gdrive')
```

```
↳ Mounted at /content/gdrive
```

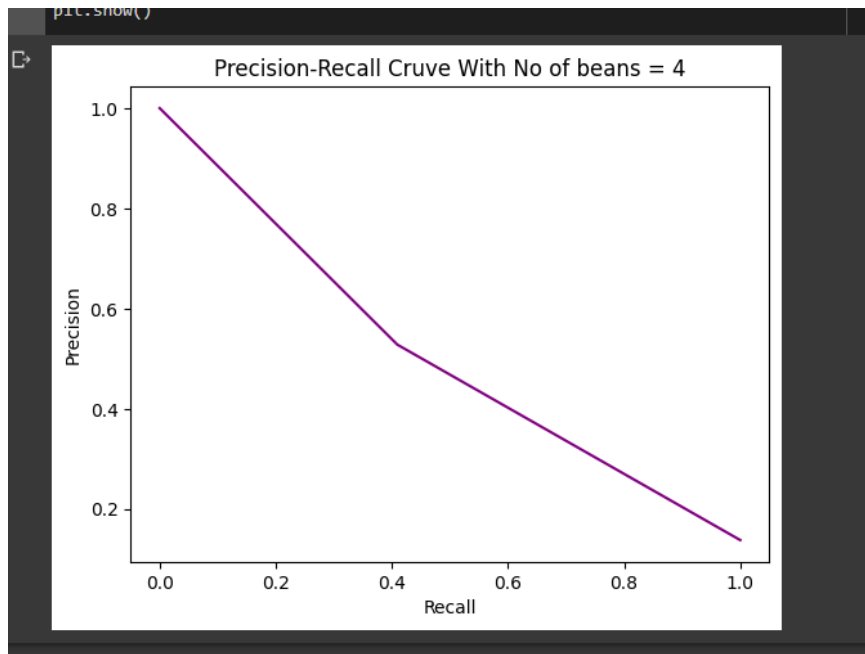
```
[ ] !unzip gdrive/My\ Drive/archive.zip
```

Naive Bayes method:

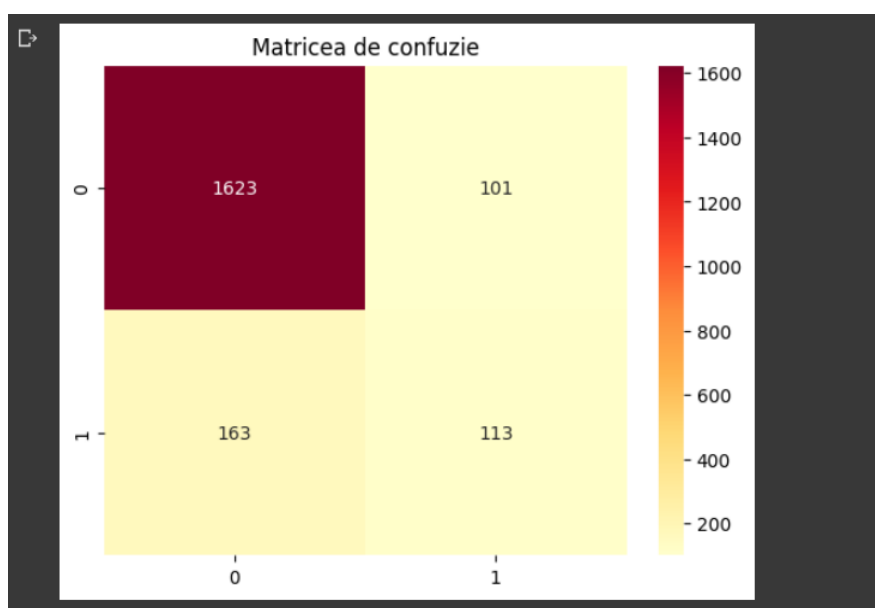
The first attempt used the MultinomialNB model from the `sklearn.neaive_bayes` library. This model uses Bayes' theorem, so initially a pre-processing of the data was needed and, more precisely, the grouping of the training images into a number of bins. Next, I will attach the plotting of some graphs to illustrate the performance of the model depending on the number of bins used (I will refer only to the interval of bins [2,10], because after several attempts it proved to be the most effective).



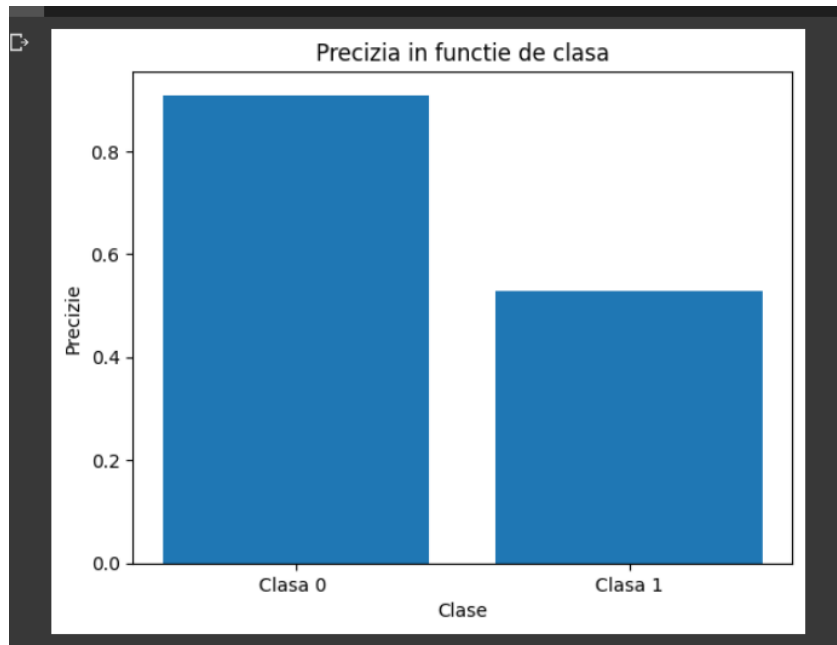
Therefore, the model with 4 bins seems to be the most efficient, having the following precision-recall curve:



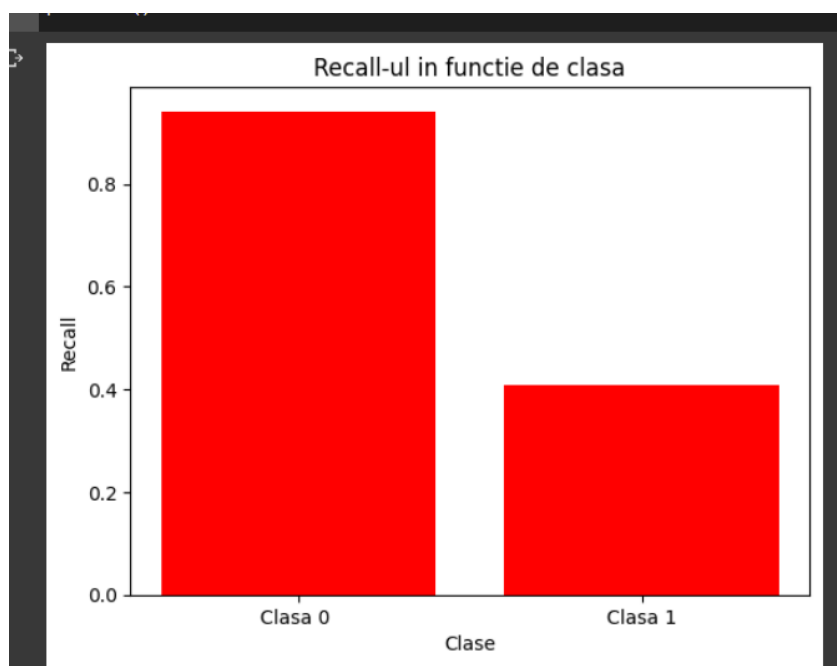
Confusion matrix:



Accuracy (for each class separately):



Recall (for each class separately):



And the accuracy of the model:

```
getAccuracyValue(conf_matrix)

0.868
```

In order to obtain these results, in addition to the universal functions defined above, we defined the method *check_best_accuracy* which receives a list containing the number of bins to try and returns a dictionary with the accuracy of each model on the validation set.

```
# functia utilizata pentru compararea modelelor
# va returna un dictionar de forma {numar_binuri: acurateatea_pe_setul_de_validare}

def check_best_accuracy(test_numbers):
    accuracy = {}
    for test_number in test_numbers:
        print(test_number)
        accuracy[test_number] = MultinomialNB_method("validation_score", test_number)
    return accuracy
```


It can be noted that the MultinomialNB_method function can be used both to obtain the score on the validation data, the predictions on the validation data, as well as the predictions on the test data, depending on the purpose with which we call it (specified in the mode parameter).

```
# metoda ce imparte poza (matricea) in numarul de binuri specificat
# intorc un csr_matrix, intrucat aritmetice sunt mult mai rapide in cadrul acestuia
def values_to_bins(matrice, bins):
    digitized = np.digitize(matrice, bins) - 1
    return csr_matrix(digitized)

# metoda ce antreneaza modelul Multinomial in functie de parametrii specificati
def MultinomialNB_method(mode, num_bins, no_to_proecess_at_a_time=500):
    # pixelii pozelor au valorile cuprinse intre 0 si 255
    # stabilesc valorile ce vor fi mapate fiecarui bean, in functie de paraetrul num_bins
    bins = np.linspace(start=0, stop=255, num=num_bins)
    # definesc modelul
    naive_bayes_model = MultinomialNB()
    # stabilesc numatul de iteratii, in functie de numarul de imagini pe care vreau sa le procesez la un moment dat
    iterations = int(np.ceil(len(train_images) / no_to_proecess_at_a_time))
    for index in range(iterations):
        # gasesc indicele de start al procesarii curente
        start_index = index * no_to_proecess_at_a_time
        # verific daca nu cumva am ajuns la finalul imaginilor de antrenament
        end_index = min((index + 1) * no_to_proecess_at_a_time, len(train_images))
        # impart imaginile de antrenament in binuri
        train_images_as_bins = values_to_bins(train_images[start_index:end_index], bins)
        # gasesc labelurile respective
        current_train_labels = train_labels[start_index:end_index]
        # antrenez modelul
        naive_bayes_model.partial_fit(train_images_as_bins, current_train_labels, classes=np.unique(train_labels))
    # in functie de modul de apelare al functiei returnez outputul dorit
    if mode == "validation_score":
        return naive_bayes_model.score(values_to_bins(validation_images, bins), validation_labels)
    elif mode == "validation_predict":
        return naive_bayes_model.predict(values_to_bins(validation_images, bins))
    else:
        return naive_bayes_model.predict(values_to_bins(test_images, bins))
```

This model (with 4 bins), which proved to be the most effective for the Naive Bayes method, obtained a score of 4.1558 in the competition.

Failed attempts + some conclusions:

- The first 3 submissions got the result 0 on the Kaggle platform, because I omitted to divide the test data into bins.
- The RAM memory exceeded the limit of 12 GB imposed by google colab, because in the function of reading the images, I retained them in an np.array, having a syntax of the form: `img = np.append(img, current_image)`. Thus we found out that the operations of adding and copying an np.array can be very slow and can consume a large amount of resources, especially if the size of the array is considerable. So I had to optimize the `get_images` function, keeping the images in a simple list, before converting them at once into an np.array. Also, in order to avoid overloading a data structure with a large number of elements, I finally reached the implementation of the logic according to which I periodically empty the list of images.
- The conclusions we reached after these tests with the Naive Bayes model (period of approx. one week) were the following:
 - 1. A more efficient algorithm for binary image classification is needed
 2. The validation + training data sets (and probably also the testing one) contain many more examples from class 0. -> therefore an approach that takes into account the unbalanced data will be needed.
 3. It is very useful to plot and analyze at least some of the training/validation/test images, in order to understand what the important features are (an aspect that I have addressed in future attempts).
 4. By understanding the usefulness of certain libraries and/or methods, I can significantly optimize the training time of my model (ie `csr_matrix`)
- After a more detailed research, I understood that the best approach for such a problem is the Convolutional Neural Network model and I decided to proceed to its implementation.

CNN:

● Version 1: Best platform score: approx 0.6

Features:

- Normalization of images
- Transformation of images from np.array into tensors
- Transforming lists of labels into tensors
- Identification of the number of images classified with "1"

● Version 2: Best platform score: approx 0.62

Features:

- Saving the best model according to certain metrics
- Creating a custom checkpoint
- Adaptive Learning Rate
- Bias predictions
- Image cropping
- Adding layers

● Version 3: Best platform score: approx 0.7

Features:

- Pipeline for oversampling + undersampling
- Combining the best predictions
- Focusing on the f1 metric

CNN - Version 1

Convolutional Neural Network is a very efficient algorithm in image classification. It consists of a neural network, having the particularity of convolutional layers that are responsible for extracting high-level features from images, then transformed into probabilities by the dense layers.

I used functions from the cv2 library to normalize the images and tensorflow to define the model + some other useful functions (ie conversion to tensors). Thus, the function that normalizes the images and transforms them into tensors is the following:

```
def normalize_image_and_convert_to_tensor(image):  
    # converteste imaginea intr un np array float32  
    img_float32 = np.float32(image)  
    # normalizeaza pixelii intre 0 si 1  
    img_normalized = cv2.normalize(img_float32, None, 0, 1.0, cv2.NORM_MINMAX)  
    # adauga o noua dimensiune pentru a transforma array-ul in format 3D  
    img_normalized_3d = np.expand_dims(img_normalized, axis=-1)  
    # converteste in tensor  
    img_tensor = tf.convert_to_tensor(img_normalized_3d)  
    return img_tensor
```

A logic error can already be observed in this version, as the images were reconverted into a 3D array, although the convert('L') method made them black and white beforehand.

I used the Sequential model from tensorflow, initially having a structure similar to this one:

```
def create_cnn_model_to_train():  
    model = models.Sequential()  
    model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(224, 224, 1)))  
    model.add(layers.MaxPooling2D((2, 2)))  
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))  
    model.add(layers.MaxPooling2D((2, 2)))  
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))  
    model.add(layers.Flatten())  
    model.add(layers.Dense(16, activation='relu'))  
    model.add(layers.Dense(16, activation='relu'))  
    model.add(layers.Dense(16, activation='relu'))  
    model.add(layers.Dense(16, activation='relu'))  
    model.add(layers.Dense(16, activation='softmax'))  
  
    return model
```

Although the obtained performances were better, several problems can be observed.

1. The last layer should have only 2 neurons, being a binary classification problem.
2. With the increase in the number of epochs, the model risks overfitting.

So that the final version, the one that I will analyze next. dropout and batchNormalization layers are used to avoid overfitting because:

- Dropout randomly removes a percentage of neurons from the previous layer.
- BatchNormalization adjusts the mean and standard deviation of the images.

Also, the last layer received the optimal size for the softmax activation function. Thus, although I added a considerable number of layers, the model no longer overfits so quickly and became more robust.

```
def create_cnn_model_to_train():
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 1)))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Dropout(0.2))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Dropout(0.2))
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Dropout(0.2))
    model.add(layers.Conv2D(256, (3, 3), activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Dropout(0.2))
    model.add(layers.Conv2D(512, (3, 3), activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Dropout(0.2))
    model.add(layers.Flatten())
    model.add(layers.Dense(512, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(256, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(2, activation='softmax'))

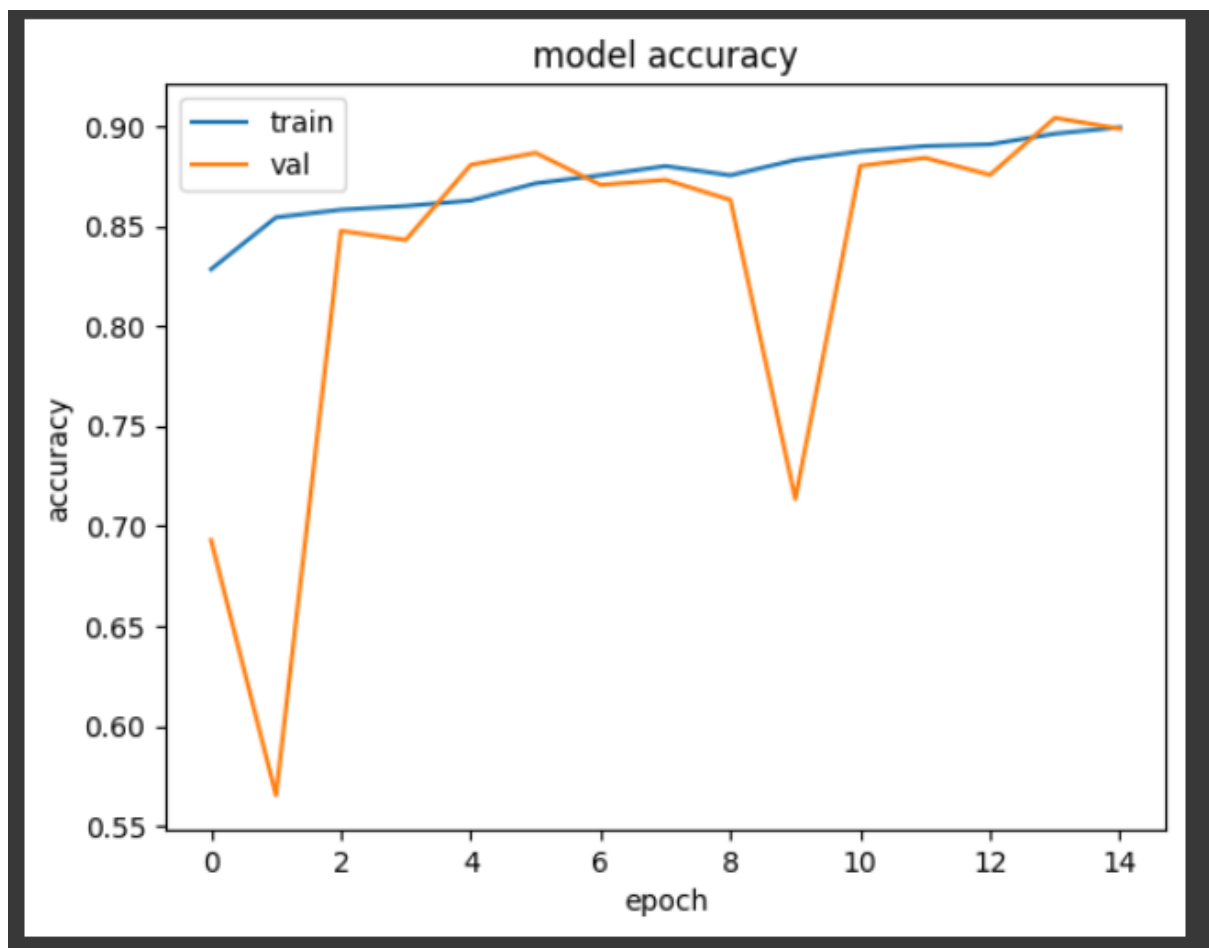
    return model
```

I encountered some difficulties in defining the optimal way to compile the model, finally arriving at the form:

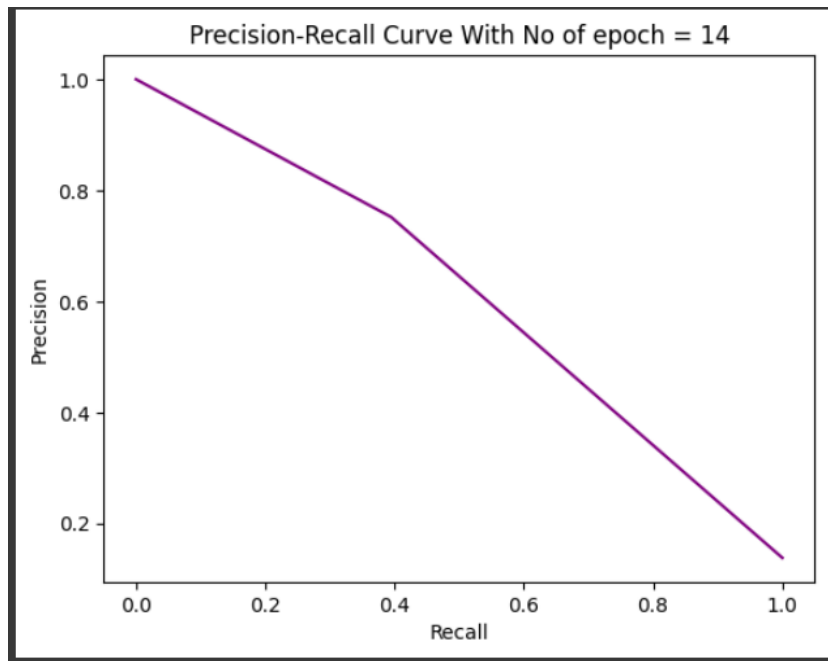
```
cnnModel.compile(optimizer = optimizers.Adam(),  
                 loss=tf.keras.losses.BinaryCrossentropy(),  
                 metrics=['accuracy'])
```

Although I would find out later that 'accuracy' is not really the most suitable metric for such a problem.

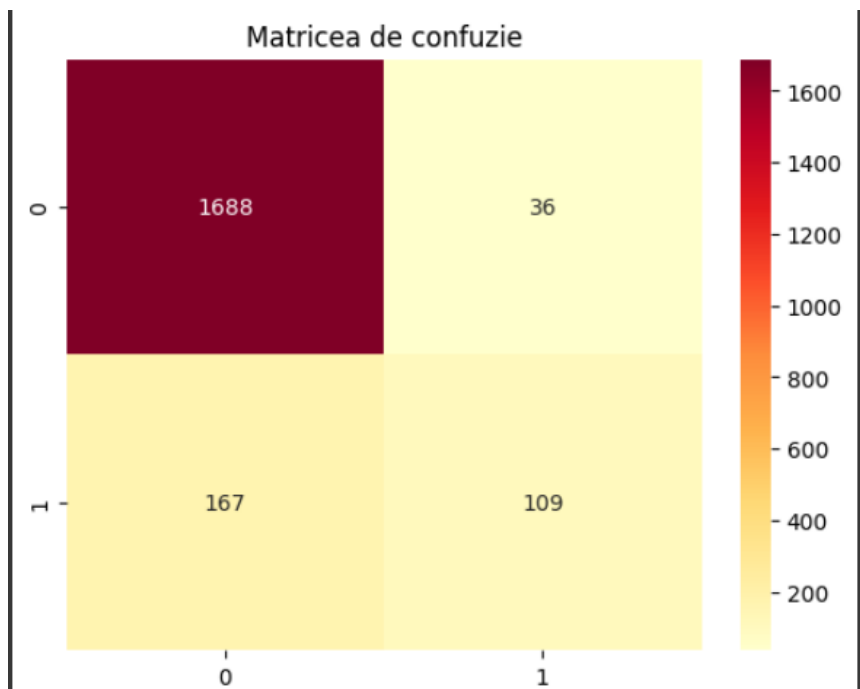
Using the graph below, I decided to train the model on the optimal number of epochs, namely: 14



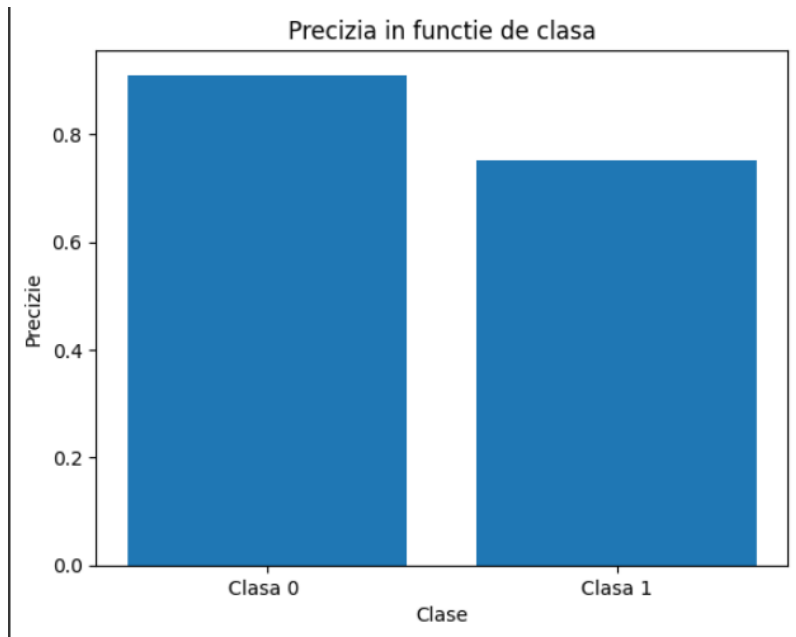
Thus, we obtained the precision-recall curve:



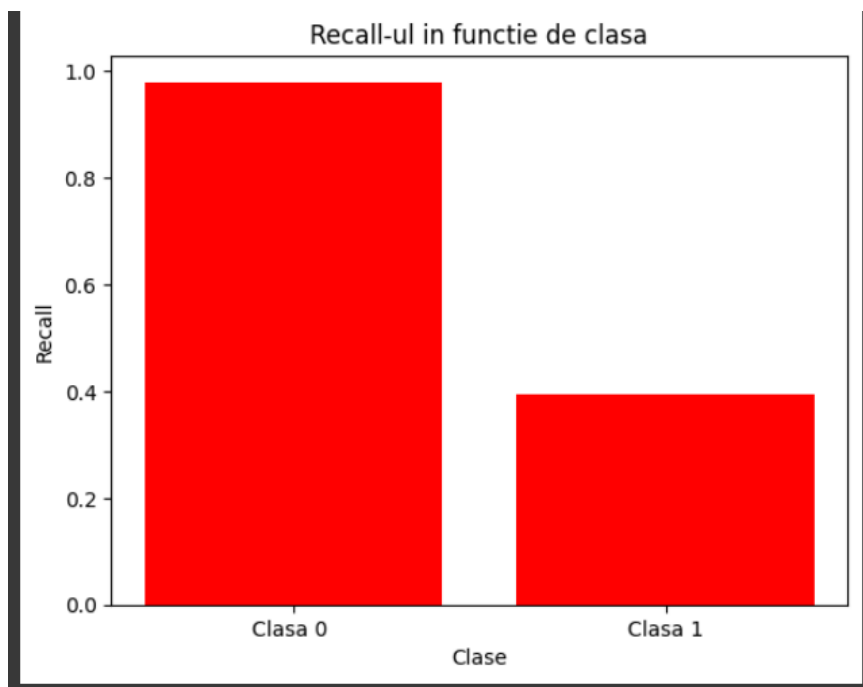
Confusion matrix:



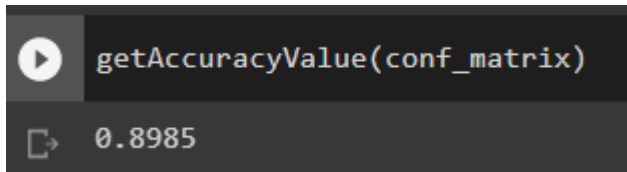
Accuracy (for each class separately):



Recall (for each class separately):



And the accuracy on the validation set:



```
getAccuracyValue(conf_matrix)
```

0.8985

Failed attempts + some conclusions:

- Although it seemed that I was doing everything correctly, the .fit method of the keras model threw me errors, because my labels were not one-hot encoding.
- I noticed a correlation between the score obtained on Kaggle and the number 1 in the output (the best had between 400 and 700 predictions 1).
- Although the accuracy of the validation data was high, the score on the platform was much lower.
- Classic data augmentation did not help, on the contrary (using techniques such as mixup)
- Instead of the repetitive structure:

```
while True:
    fit_details = cnnModel.fit(train_images, train_labels, epochs=1 ,
                               validation_data=(validation_images, validation_labels), shuffle=True)
    predictions=cnnModel.predict(test_images)
    classes_predictions=np.argmax(predictions,axis=1)
    results = cnnModel.evaluate(validation_images, validation_labels, batch_size=100)
    no_of_1 = np.count_nonzero(classes_predictions)
    print(no_of_1)
    if no_of_1 > 530 and no_of_1 <650:
        break
```

I could use a checkpoint to save the best model.

CNN - Version 2

In addition to the optimizations brought in version 1, I added a custom checkpoint to the fit function that displays the confusion matrix and saves the model with the least 1 misclassified, if the accuracy on the validation data was over 90. For this, I overwrote the `on_epoch_end` method.

```
def on_epoch_end(self, epoch, logs=None):
    global no_of_ones
    global max_f1_score
    predictions = self.model.predict(validation_images)
    accuracy = self.model.evaluate(validation_images, validation_labels)[1]
    #uncomment for bias predict
    #conf_matrix = confusion_matrix([np.argmax(pred) for pred in validation_labels], self.get_bias_class_predict(predictions))
    classes_predictions=[np.argmax(pred) for pred in predictions]
    conf_matrix = confusion_matrix([np.argmax(pred) for pred in validation_labels], classes_predictions)
    print(conf_matrix)
    current_f1 = f1_score([np.argmax(pred) for pred in validation_labels], classes_predictions)
    if conf_matrix[1][0] <= no_of_ones and accuracy > 0.90:
        no_of_ones = conf_matrix[1][0]
        self.model.save(self.filepath, overwrite=True)
        print(f'Saved model at epoch {epoch+1} with no_of_ones_missclassified={no_of_ones}')
```

I also added the `get_bias_predict` function, which classifies my images as 1, if they had a probability of over 0.4 (standard being 0.5 for softmax). I did this in an attempt to force the model to classify more of my images as having label 1, since at this stage I had problems with this aspect.

```
def get_bias_class_predict(self, predictions):
    bias_predict = []
    for pred in predictions:
        if pred[1] >= 0.4:
            bias_predict.append(1)
        else:
            bias_predict.append(0)
    return bias_predict
```

Besides this Custom callback, I added a learning rate callback, which halved the learning rate if the val_loss of the model increased consecutively 2 epochs in a row, compiling the model with an initial learning_rate of 0.0007.

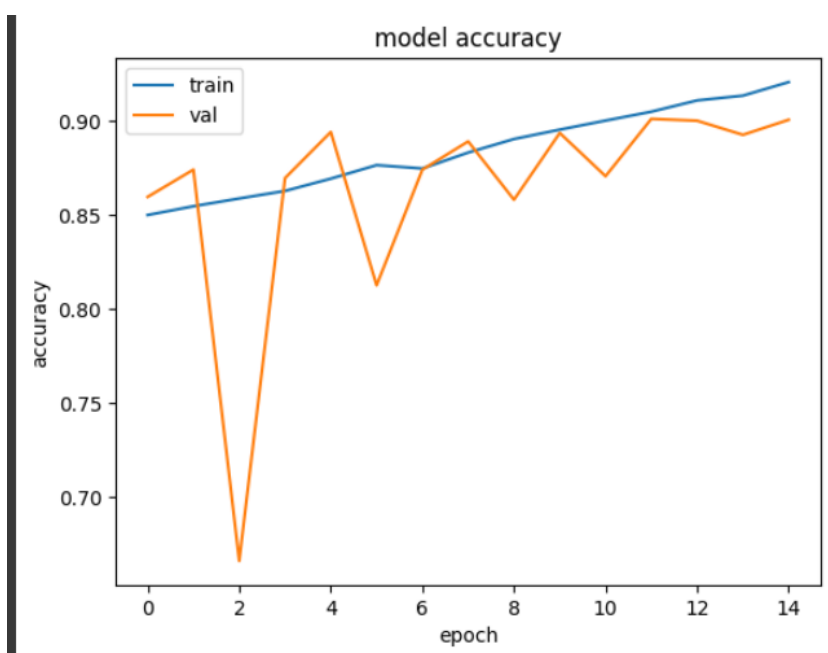
The last callback added was the one that stopped my fit process if val_loss had not improved for 4 epochs.

```
cnnModel = create_cnn_model_to_train()
adaptable_learning_rate = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2, verbose=1)
early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=4)
```

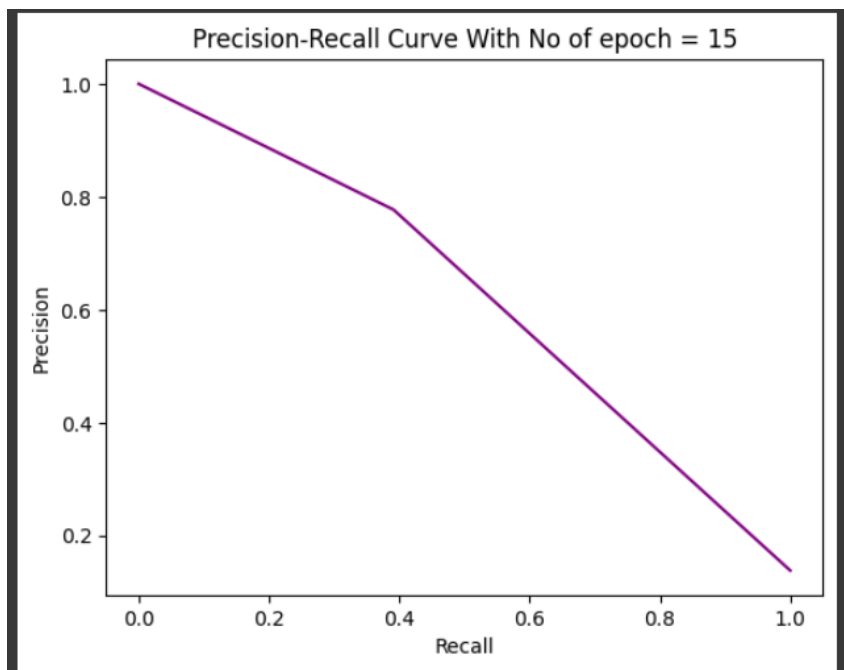
The last major change compared to the first version was the fact that I cut the training, testing and validation images to get rid of the black edges:

```
def normalize_and_cut_image(image):
    # convert to float32
    img_float32 = np.float32(image)
    #cutting the image
    img_float32 = img_float32[[18:200]]
    img_float32 = img_float32[:,20:200]
    # normalize to the range 0-1
    img_normalized = cv2.normalize(img_float32, None, 0, 1.0, cv2.NORM_MINMAX)
    return img_normalized
```

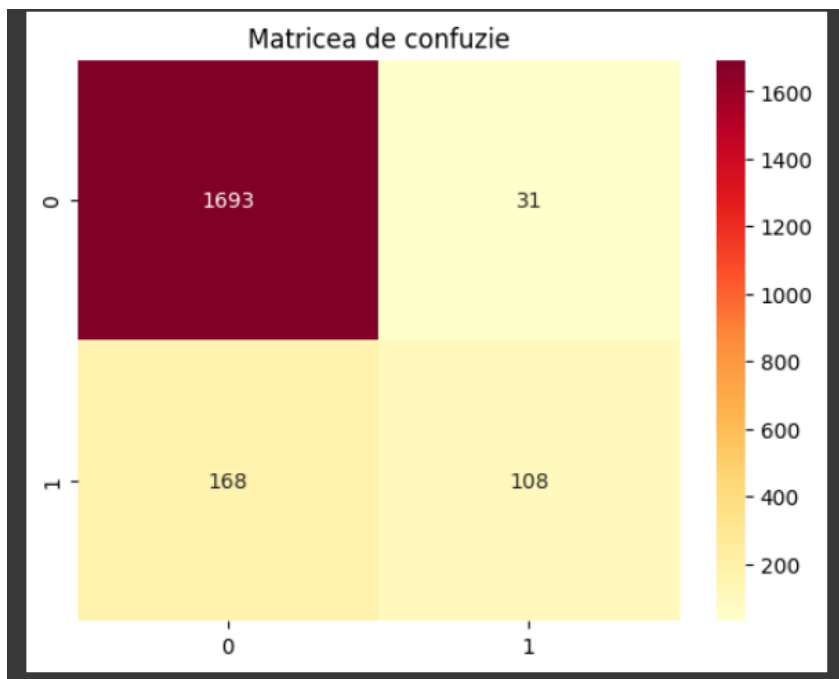
For 15 epochs we obtained the graph:



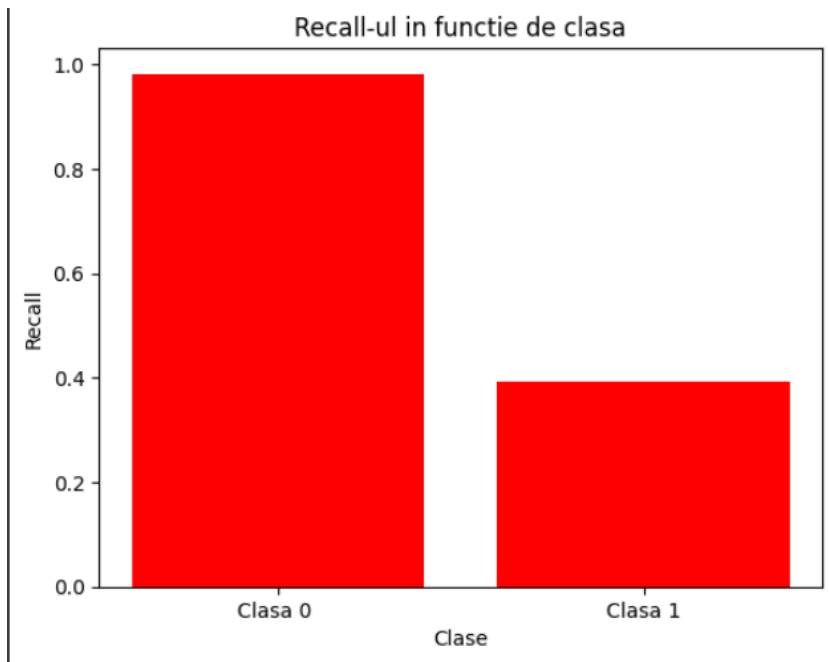
Precision-recall curve:



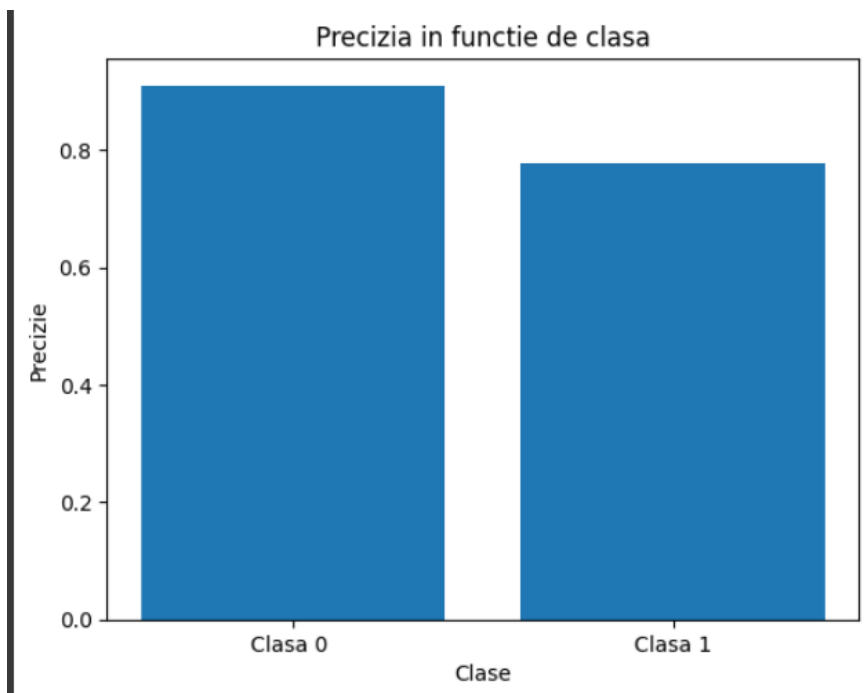
Confusion matrix:



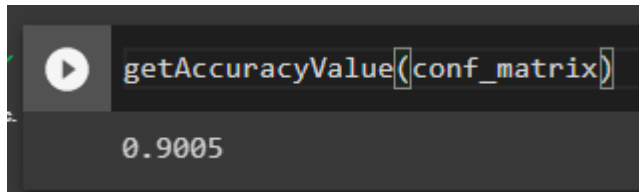
Recall (for each class separately):



Accuracy (for each class separately):



Accuracy on the validation data set:



```
getAccuracyValue(conf_matrix)
```

0.9005

Failed attempts + some conclusions:

- Until I switch to an execution with more GPU RAM available, the google colab session kept getting interrupted because the custom callbacks are much more inefficient and memory consuming than those defined by tensorflow.
- I had to somehow manage to train the models so that they predict more than 1 correctly.
- The variant of get_bias_predict does not work as well as I expected.

CNN – Version 3

One of the biggest benefits of this version was represented by the function *get_average_predictions*, which received as a parameter a list of files in csv format and returned a new list of predictions according to the probability of each class (ie if image 100 was cataloged by 3 models as 1 and by 2 as 0, then the output would have been 1).

```
def get_average_predictions(files_list):
    val_list = [0 for i in range (0, 5149)]
    new_predict=[None for i in range (0, 5149)]
    for filename in files_list:
        with open(filename, 'r') as f:
            f.readline()
            for i in range (0, 5149):
                number,prediction = [int(x) for x in f.readline().split(",")]
                if prediction == 0:
                    val_list[i] -= 1
                else:
                    val_list[i] += 1
            f.close()
    for index in range (0, 5149):
        print (val_list[index])
        if val_list[index] < 0:
            new_predict[index] = 0
        else:
            new_predict[index] = 1
    return new_predict
```

Thus, we were able to combine the best models saved with the help of the checkpoint and the score on the platform increased significantly.

Another difference was the implementation of a pipeline using the smote algorithm of undersampling for class 0 and oversampling for class 1:

```
add_minority_samples = SMOTE(random_state=38,sampling_strategy=0.8)
delete_majority_samples = RandomUnderSampler()
sampling_strategy = [('o', add_minority_samples), ('u', delete_majority_samples)]
apply_sampling = Pipeline(steps=sampling_strategy)

# redimensionăm imaginile într-un tensor unidimensional pentru a putea fi utilizat cu SMOTE
train_images_resaped = train_images.reshape((-1, 120*120))

# aplicăm SMOTE pe setul de date
train_images, train_labels = apply_sampling.fit_resample(train_images_resaped, train_labels)

# redimensionăm imaginile înapoi în forma inițială
train_images = train_images.reshape((-1, 120, 120))
```

Thus the test data set became:

```
from collections import Counter
print(Counter(train_labels))
print(train_images.shape)

Counter({0: 10209, 1: 10209})
```

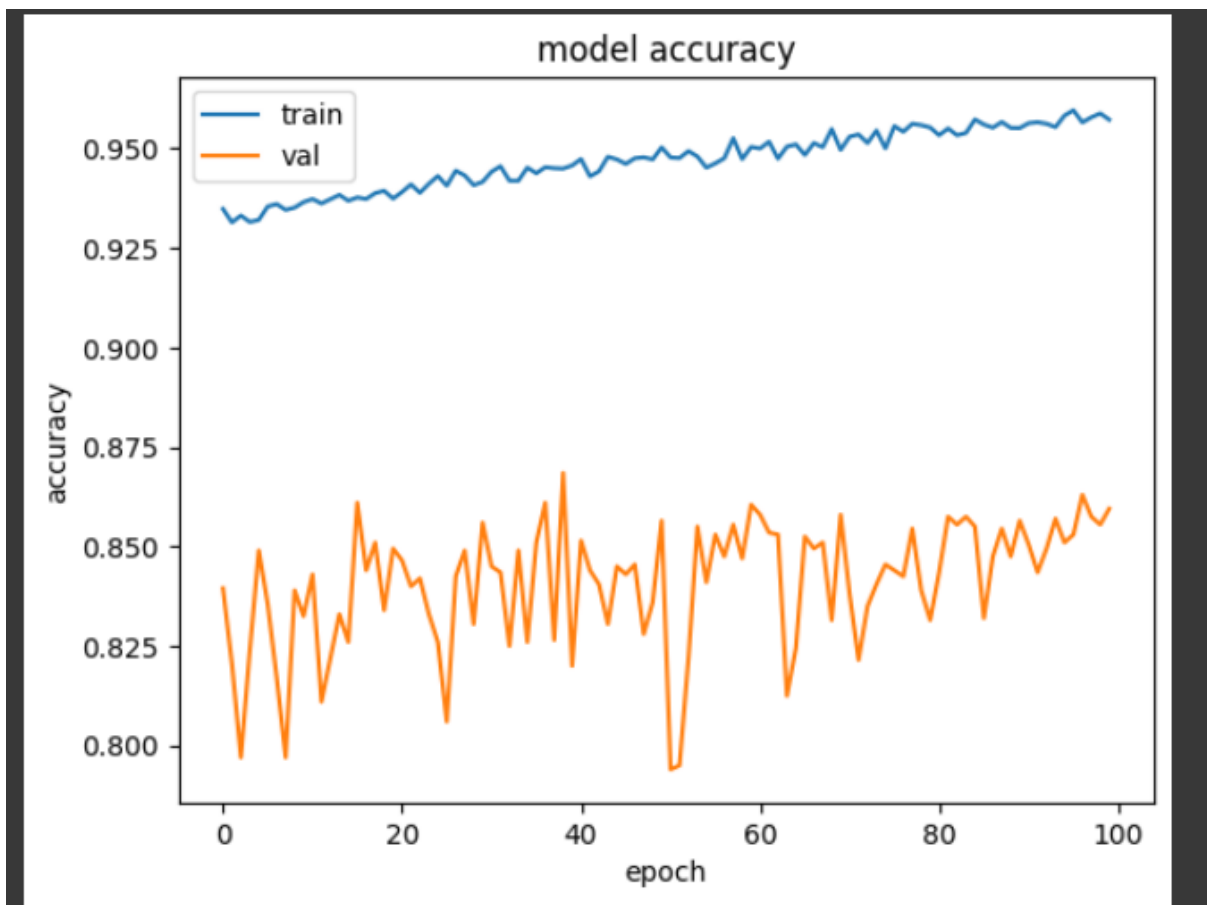
In this approach, I used the f1 metric to save the best models:

```
def on_epoch_end(self, epoch, logs=None):
    global no_of_ones
    global max_f1_score
    predictions = self.model.predict(validation_images)
    accuracy = self.model.evaluate(validation_images, validation_labels)[1]
    #uncomment for bias predict
    #conf_matrix = confusion_matrix([np.argmax(pred) for pred in validation_labels], self.g
    classes_predictions=[np.argmax(pred) for pred in predictions]
    conf_matrix = confusion_matrix([np.argmax(pred) for pred in validation_labels], classes
    print(conf_matrix)
    current_f1 = f1_score([np.argmax(pred) for pred in validation_labels],classes_predictio
    if current_f1 > max_f1_score:
        max_f1_score = current_f1
        self.model.save(self.filepath, overwrite=True)
        print(f'Saved model at epoch {epoch+1} with f1_score={max_f1_score}')
```

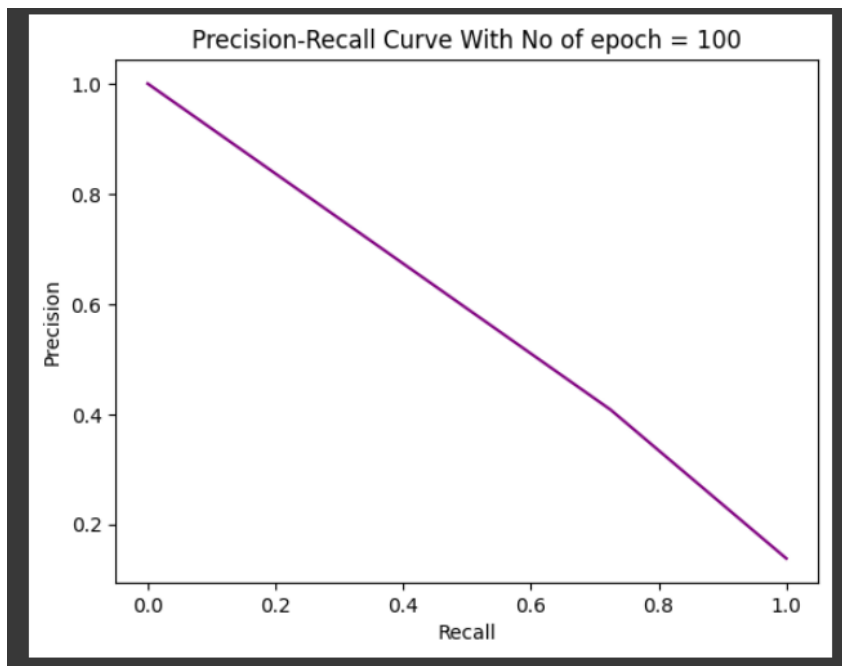

I also noticed that if you cut the pictures even more, their important attributes are not lost, and the learning time for each epoch improved from 80 seconds to 10 seconds.

```
img_float32 = img_float32[50:170]  
img_float32 = img_float32[:,50:170]  
# normalize to the range 0-1
```

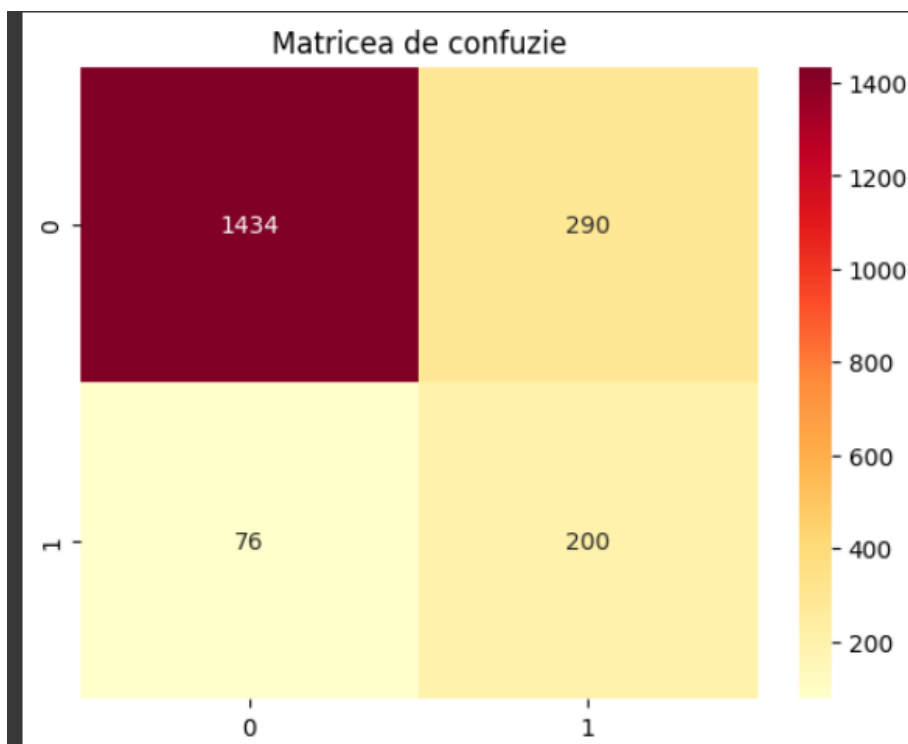
We obtained the following performances depending on the era:



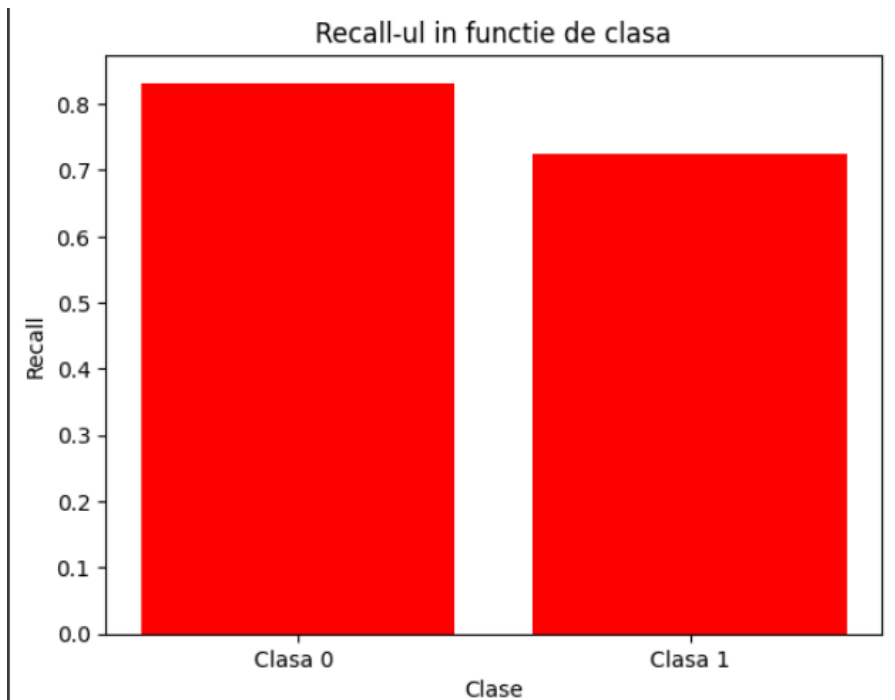
Precision-recall curve:



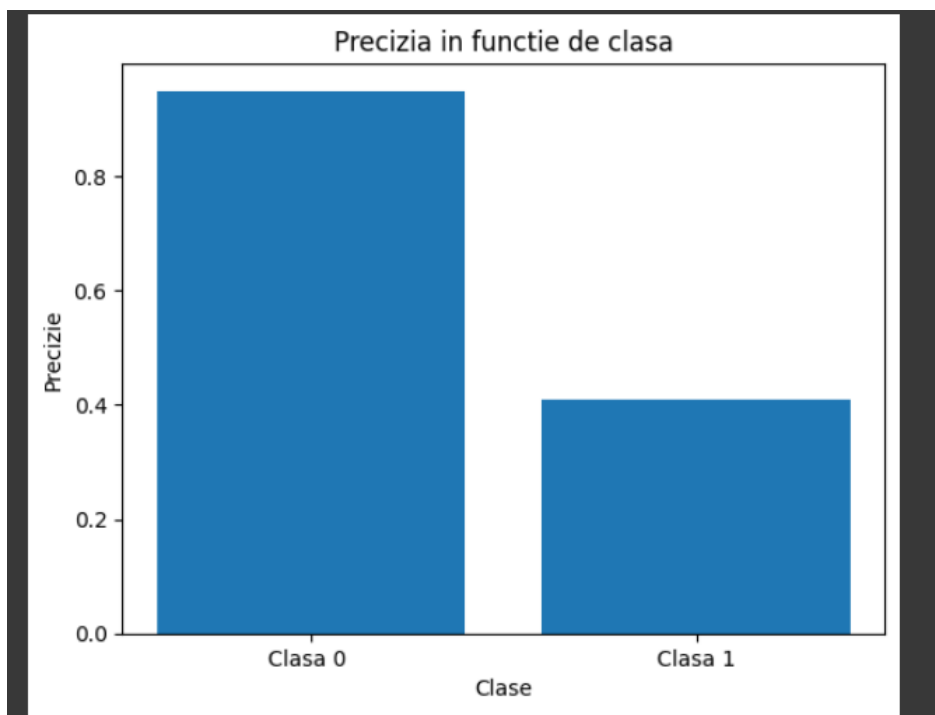
Confusion matrix:



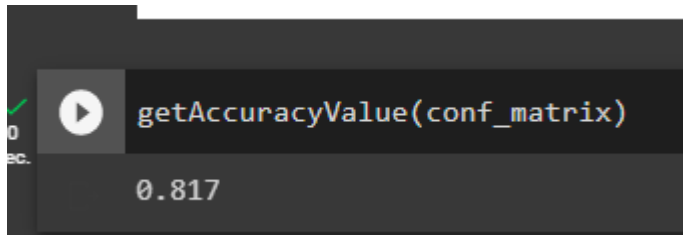
Recall (for each class separately):



Accuracy (for each class separately):



Accuracy:

A screenshot of a Jupyter Notebook cell. The cell contains the code `getAccuracyValue(conf_matrix)`. To the left of the code is a play button icon. Below the code, the output `0.817` is displayed. On the far left, there is a green checkmark and some partially visible text: "0" and "ec.".

```
0 ✓  
ec. [ ] getAccuracyValue(conf_matrix)  
      0.817
```

Failed attempts + some conclusions:

- Oversampling is very useful, but the model consumes much more resources to make it fit
- In such cases, undersampling can also help
- It is a good idea to combine the best models I had
- In classification problems with unbalanced data, the accuracy metric is not the most appropriate