

Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**RAPORT**

**Lucrarea de laborator nr.5**

**la Inteligența Artificială**

*Tema: Rețele neuronale artificiale*

Grupa academică: TI-211  
A efectuat: Popa Cătălin

**A verificat: Mariana Rusu**

Chișinău 2024

Rețelele neurale artificiale (RNA) sunt modele matematice și computaționale inspirate de funcționarea creierului uman, care sunt utilizate în învățarea automată și în domenii precum recunoașterea de imagini, recunoașterea vocală, traducerea automată și multe altele. Acestea sunt alcătuite dintr-o serie de straturi (numite și straturi de neuroni) care procesează informațiile și aplică operații matematice pentru a face predicții sau clasificări.

**Neuronii artificiali**: Sunt unități de bază ale unei rețele neuronale artificiale, care îndeplinesc funcția analogă neuronilor biologici. Acești neuroni primesc intrări, aplică o anumită operație matematică asupra acestora și produc o ieșire.

**Straturi**: Rețelele neurale sunt alcătuite din mai multe straturi de neuroni. Cele mai comune tipuri de straturi sunt:

* Straturi de intrare: Primele straturi ale rețelei, care primește datele de intrare.
* Straturi ascunse: Straturi intermediare între straturile de intrare și straturile de ieșire. Acestea procesează și extrag caracteristici din date.
* Straturi de ieșire: Ultimul strat al rețelei, care produce ieșirea finală.

**Conexiuni ponderate:** Conexiunile între neuroni sunt asociate cu ponderi care influențează importanța intrării lor. În timpul antrenamentului, aceste ponderi sunt ajustate pentru a îmbunătăți performanța modelului.

**Funcții de activare:** Fiecare neuron aplică o funcție de activare asupra sumei ponderate a intrărilor sale pentru a produce ieșirea. Funcțiile de activare pot fi sigmoid, tanh, ReLU etc.

**Antrenare și propagare înapoi:** Antrenarea rețelelor neurale implică ajustarea ponderilor conexiunilor pentru a minimiza o funcție de pierdere. Acest lucru este realizat folosind algoritmi de optimizare și propagare înapoi, care calculează gradientul funcției de pierdere și ajustează ponderile în consecință.

**Arhitecturi complexe:** Există numeroase arhitecturi de rețele neurale, precum rețelele feedforward, rețelele convoluționale (CNN), rețelele recurente (RNN), rețelele neurale cu memorie a lungă și scurtă (LSTM), rețelele generative adversar (GAN), etc., care sunt adaptate pentru diferite sarcini și tipuri de date.

În ansamblu, rețelele neurale artificiale sunt un instrument puternic în învățarea automată, capabil să învețe și să extragă modele complexe din date și să rezolve o varietate de probleme.

În primul pas, am important arhiva cu toate imaginile. Am două directorii, unul cu mașini și altul cu motociclete.

!unzip -q "/content/Vehicles2.zip"

!ls "Vehicles/"

Apoi, am importat bibliotecile necesare.

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import pathlib

import os

from tqdm import tqdm

Afișăm numărul de imagini cu mașini și numărul de imagini cu motociclete.

len(os.listdir("Vehicles/Cars"))

414

len(os.listdir("Vehicles/Moto"))

425

Creăm all\_paths, care va conține toate căile către fișierele .jpg din subdirectoarele din directoriu Vehicles. Aceste căi vor fi utilizate mai târziu pentru a citi și prelucra aceste fișiere în continuare.

DataPath = pathlib.Path("Vehicles")

all\_paths = DataPath.glob("\*/\*.jpg")

all\_paths

<generator object Path.glob at 0x7d45ed8f76f0>

Transformăm all\_paths într-o listă de căi către fișierele .jpg. După voi afișa primele 10 căi utilizând segmentarea listei [0:10], care va returna primele 10 elemente din listă.

all\_paths = list(all\_paths)

all\_paths[:10]

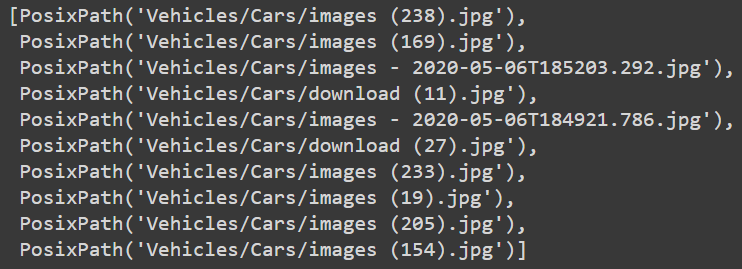


Figura 1 -Afișare primele 10 căi

Convertim căile din obiecte de tip Path în șiruri de caractere.

all\_paths = list(map(lambda x : str(x), all\_paths))

all\_paths[:10]

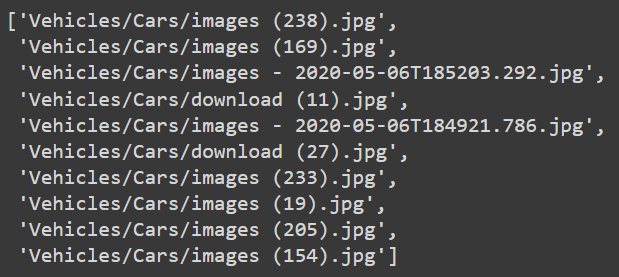


Figura 2 – Convertirea în șiruri de caractere

Amestecăm ordinea elementelor din lista all\_paths, folosind funcția shuffle() din modulul random.

Astfel, ordinea căilor către fișierele jpg din directoriul Vehicle va deveni aleatorie.

from random import shuffle

shuffle(all\_paths)

all\_paths[:10]

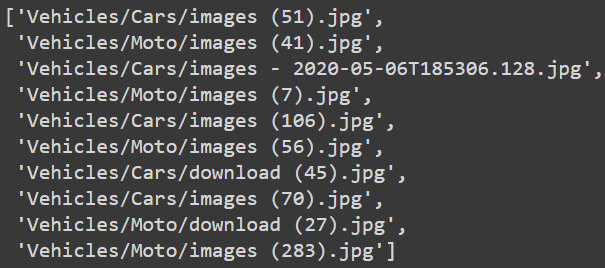


Figura 3 – Amestecarea path-urilor

În continuare, vom decoda imaginile. Funcția TestImageQuality(app\_paths) primește o listă de căi către fișierele de imagini și verifică calitatea accestora.

def TestImageQuuality(all\_paths):

  new\_all\_paths = []

  for path in tqdm(all\_paths):

    try:

      image = tf.io.read\_file(path)

      image = tf.io.decode\_jpeg(image, channels=3)

    except:

      continue

    new\_all\_paths.append(path)

  return new\_all\_paths

all\_paths = TestImageQuuality(all\_paths)

all\_paths[:10]

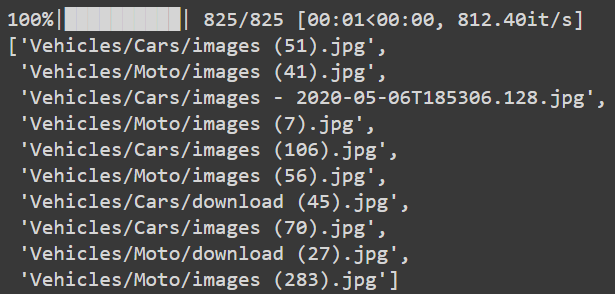


Figura 4 – Decodarea imaginilor

Creăm funcția get\_label(image\_path) care primește calea către un fișie de imagini și returnează eticheta corespunzătoare. Folosim funcția split(/) pentru a separa calea către fișierul de imagine în elemente individuale bazate pe separatorul /.

def get\_label(image\_path):

  return image\_path.split("/")[-2]

all\_labels = list(map(lambda x: get\_label(x), all\_paths))

all\_paths[:10]

În următoarea secțiune, am utilizat clasa LabelEncoder din biblioteca scikit-learn pentru a transforma etichetele claselor într-o formă numerică, astfel încât să poartă fi utilizate mai ușor în algoritmi de învățare automată.

Se crează un obiect LabelEncoder denumit Le. Se aplică metodă fit\_transform() a obiectului Le pe lista all\_labels. Această metodă ajustează encoderul la etichetele din all\_labels și le transformă într-o reprezentare numerică. Rezultatul transformării este stocat înapoi în all\_labels. Se afișează primele 10 elemente din all\_labels, pentru a examina etichetele transformate numeric. Acestea sunt acum reprezentările numerice ale etichetelor claselor, care sunt utilizate adesea în algoritmi de învățare automată.

from sklearn.preprocessing import LabelEncoder

Le = LabelEncoder()

all\_labels = Le.fit\_transform(all\_labels)

all\_labels[:10]



Figura 5 – Afișarea etichetelor transformate în 0 și 1

Acum, vom crea setul de antrenamente. Aici se folosește funcția train\_test\_split() din biblioteca scikit-learn pentru a împărți datele într-un set de antrenare și un set de testare. Se furnizează argumentele all\_paths și all\_labels către funcția train\_test\_split(). Acestea reprezintă datele de intrare (căile către fișierele de imagine) și etichetele corespunzătoare. Funcția train\_test\_split() va împărți aceste date în patru variabile separate: Train\_paths (căile către fișierele de imagine pentru setul de antrenare), Val\_paths (căile către fișierele de imagine pentru setul de testare), Train\_labels (etichetele corespunzătoare pentru setul de antrenare) și Val\_labels (etichetele corespunzătoare pentru setul de testare). Setul de antrenare este folosit pentru a antrena modelul, în timp ce setul de testare este utilizat pentru a evalua performanța modelului pe datele necunoscute. Se afișează primele 10 căi și etichete din setul de antrenare (Train\_paths[:10] și Train\_labels[:10]), pentru a verifica corectitudinea împărțirii datelor.

from sklearn.model\_selection import train\_test\_split

Train\_paths, Val\_paths, Train\_labels, Val\_labels = train\_test\_split(all\_paths, all\_labels)

Train\_paths[:10], Train\_labels[:10]

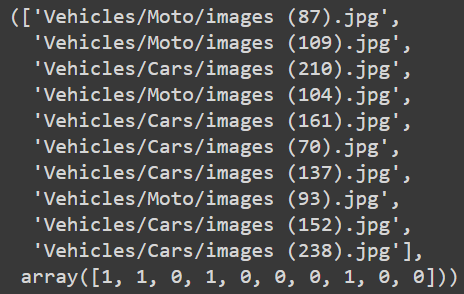


Figura 6 – Crearea setului de antrenamente

În această secțiune, este definită funcția load(image, label) care încarcă o imagine de la calea specificată și îi atașează o etichetă corespunzătoare. Această funcție este folosită pentru a încărca datele în timpul procesului de pregătire a setului de date pentru antrenarea unui model de învățare automată.

def load(image, label):

  image = tf.io.read\_file(image)

  image = tf.io.decode\_jpeg(image,channels = 3)

  return image, label

Definirea IMAGE SIZE și BATCH SIZE.

**Secvența de redimensionare (resize):**

* Utilizează tf.keras.layers.experimental.preprocessing.Resizing(IMG\_SIZE, BATCH\_SIZE) pentru a redimensiona imaginile la o dimensiune specificată.
* IMG\_SIZE specifică dimensiunea dorită a imaginilor redimensionate (înălțimea și lățimea).
* BATCH\_SIZE specifică dimensiunea lotului de imagini care va fi procesat în același timp.

IMG\_SIZE = 224

BATCH\_SIZE = 128

resize = tf.keras.Sequential([

    tf.keras.layers.experimental.preprocessing.Resizing(IMG\_SIZE, BATCH\_SIZE)

])

**Secvența de augmentare a datelor (data\_argumentation):**

* Utilizează mai multe straturi de preprocesare, cum ar fi RandomFlip, RandomRotation și RandomZoom, pentru a aplica diverse transformări și augmentări asupra datelor de intrare.
* RandomFlip("horizontal") efectuează o oglindire aleatoare a imaginilor pe orizontală.
* RandomRotation(0.2) aplică o rotație aleatoare de până la 20 de grade asupra imaginilor.
* RandomZoom(height\_factor=(-0.3, -0.2)) aplică o zoomare aleatoare asupra imaginilor, modificând înălțimea acestora cu un factor aleator între -0.3 și -0.2.
* data\_argumentation = tf.keras.Sequential([
* tf.keras.layers.experimental.preprocessing.RandomFlip("horizontal"),
* tf.keras.layers.experimental.preprocessing.RandomRotation(0.2),
* tf.keras.layers.experimental.preprocessing.RandomZoom(height\_factor=(-0.3, -0.2))
* ])

În continuare, se definește funcția get\_dataset(paths, labels, train=True) care construiește un obiect Dataset TensorFlow din căile către fișierele de imagine și etichetele corespunzătoare. Această funcție este utilizată pentru a pregăti datele pentru antrenarea unui model de învățare automată.

1. Se definește constanta AUTOTUNE pentru a specifica utilizarea optimizării automate TensorFlow pentru procesarea paralelă a datelor.
2. Funcția primește trei argumente: paths (căile către fișierele de imagine), labels (etichetele corespunzătoare) și un argument opțional train care indică dacă setul de date este pentru antrenare sau nu.
3. Se convertesc căile și etichetele în tensori TensorFlow utilizând tf.convert\_to\_tensor().
4. Se construiește un obiect Dataset pentru imagini și un obiect Dataset pentru etichete folosind tf.data.Dataset.from\_tensor\_slices().
5. Se creează un obiect Dataset combinat, asociind fiecare imagine cu eticheta sa folosind tf.data.Dataset.zip().
6. Se aplică funcția load() asupra fiecărui exemplu din setul de date folosind dataset.map(). Aceasta încarcă imaginile și le atașează etichetele corespunzătoare.
7. Se redimensionează imaginile la dimensiunea specificată utilizând secvența de redimensionare definită anterior (resize) prin aplicarea funcției resize() asupra setului de date.
8. Setul de date este amestecat utilizând dataset.shuffle(1000) pentru a îmbunătăți generalizarea modelului.
9. Setul de date este împărțit în loturi de dimensiune specificată (BATCH\_SIZE) utilizând dataset.batch(BATCH\_SIZE).
10. Dacă argumentul train este True, atunci se aplică augmentarea datelor utilizând secvența de augmentare definită anterior (data\_argumentation) prin aplicarea funcției data\_argumentation() asupra setului de date și se repetă setul de date folosind dataset.repeat().
11. Funcția returnează obiectul Dataset pregătit pentru antrenare.
12. Se măsoară timpul de execuție al funcției de pregătire a setului de date folosind %time.
13. Se extrage un lot de date de antrenament folosind next(iter(train\_dataset)).
14. Se afișează forma imaginilor și etichetelor din lotul de date.

AUTOTUNE = tf.data.experimental.AUTOTUNE

def get\_dataset(paths, labels, train=True):

  image\_paths = tf.convert\_to\_tensor(paths)

  label\_paths = tf.convert\_to\_tensor(labels)

  image\_dataset = tf.data.Dataset.from\_tensor\_slices(image\_paths)

  label\_dataset = tf.data.Dataset.from\_tensor\_slices(label\_paths)

  dataset = tf.data.Dataset.zip((image\_dataset, label\_dataset))

  dataset = dataset.map(lambda image, label: load(image, label))

  dataset = dataset.map(lambda image, label: (resize(image), label), num\_parallel\_calls=AUTOTUNE)

  dataset = dataset.shuffle(1000)

  dataset = dataset.batch(BATCH\_SIZE)

  if train:

    dataset = dataset.map(lambda image, label: (data\_argumentation(image), label), num\_parallel\_calls=AUTOTUNE)

    dataset = dataset.repeat()

  return dataset

%time train\_dataset = get\_dataset(Train\_paths, Train\_labels)

image, label = next(iter(train\_dataset))

print(image.shape)

print(label.shape)

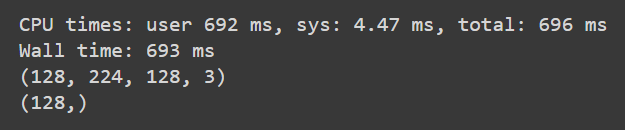


Figura 7 – Forma etichetelor și imaginilor

* 128 de imagini există într-un lot
* Fiecare imagine are o rezoluție de 224x128 și 3 canale de culoare (RGB)
* Forma etichetelor este 128, ceea ce indică că există 128 de etichete asociate cu fiecare imagine din lot

Vizualizarea a câte un exemplu din fiecare clasă mașini și motociclete(din setul de antrenamente) pentru a ne asigura că este citit corect.

print(Le.inverse\_transform(label)[0])

plt.imshow((image[0].numpy()/255).reshape(224, 128, 3))

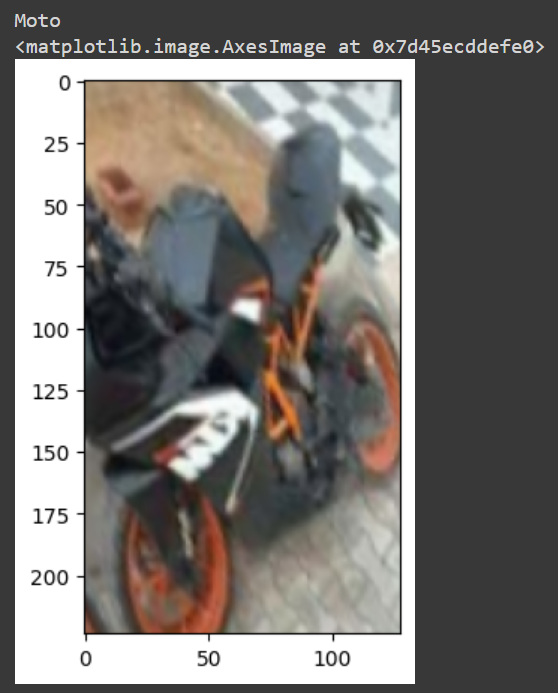


Figura 8 – Motocicletă

Aplicăm același proces ca și pentru setul de date de antrenamente, dar pentru setul de date de validatre.

%time val\_dataset = get\_dataset(Val\_paths, Val\_labels, train = False)

image, label = next(iter(val\_dataset))

print(image.shape)

print(label.shape)

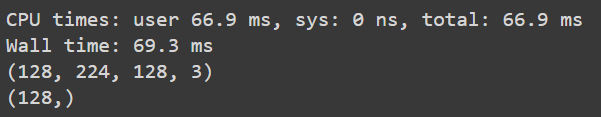


Figura 9 – Informații rulare

Afișăm datele.

print(Le.inverse\_transform(label)[0])

plt.imshow((image[0].numpy()/255).reshape(224, 128, 3))

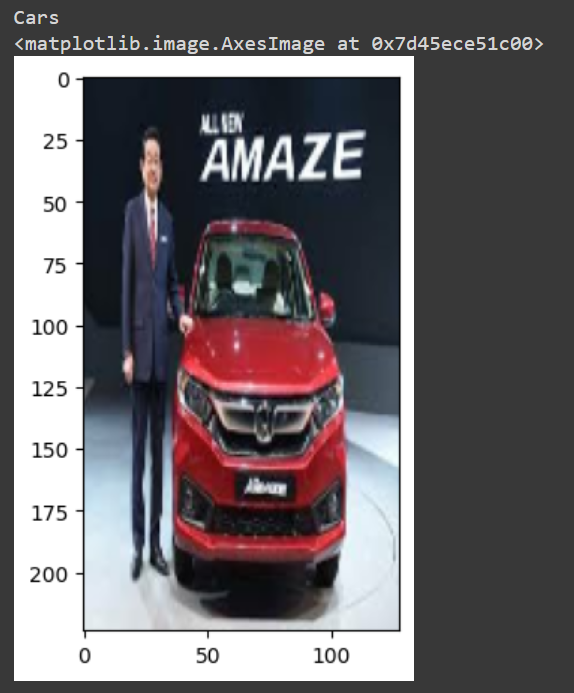


Figura 10 – Mașină

**Crearea modelului rețelei neuroonale convenționale.**

Aici se definește arhitectura unui model de rețea neural convoluțională (CNN) folosind Keras, cu următoarele caracteristici:

* Se utilizează un model secvențial (Sequential) pentru a defini modelul.
* Sunt definite patru blocuri de convoluție, fiecare urmat de o activare ReLU, normalizare în batch, strat de max-pooling și dropout pentru a preveni supradaptarea. Aceste blocuri sunt proiectate pentru a extrage caracteristici la niveluri din ce în ce mai abstracte din imaginile de intrare.
* La sfârșitul ultimului bloc de convoluție, se adaugă un strat de global max-pooling pentru a reduce dimensiunile caracteristicilor extrase.
* După stratul de global max-pooling, se adaugă două straturi complet conectate (Dense), cu activarea ReLU și dropout, pentru a procesa caracteristicile extrase și a le transforma într-o singură valoare de ieșire.
* Ultimul strat este un strat complet conectat cu o singură unitate și o activare sigmoid, utilizat pentru clasificare binară.

Această arhitectură de rețea neurală convoluțională este proiectată pentru a fi utilizată în problema specifică de clasificare a vehiculelor. Arhitectura este configurată pentru a extrage caracteristici semnificative din imaginile vehiculelor și pentru a le utiliza pentru a face predicții cu privire la apartenența acestora la una dintre cele două clase posibile.

**Vizualizarea informații despre modelul creat**

print(model.summary())

**Arhitectura modelului:**

* Modelul este o rețea neurală convoluțională (CNN) cu o structură secvențială.
* Este alcătuit dintr-o serie de straturi convoluționale, urmate de funcții de activare ReLU, normalizare în batch, straturi de max-pooling și dropout, care sunt utilizate pentru a extrage și a procesa caracteristici din imaginile de intrare.
* Modelul are un strat final de global max-pooling care reduce dimensiunea caracteristicilor extrase la un singur vector.
* Urmează două straturi complet conectate (Dense) cu activare ReLU și dropout, care sunt utilizate pentru a procesa vectorul de caracteristici extrase și a-l transforma într-o singură valoare de ieșire.
* Stratul final este un strat complet conectat cu o singură unitate și o activare sigmoid, utilizat pentru clasificare binară.

**Numărul total de parametri:**

* Modelul are un număr total de 492,993 de parametri (aproximativ 1.88 MB).
* Dintre acești parametri, 492,033 sunt parametri trainabile, iar restul de 960 sunt parametri non-trainabile (probabil parametri de normalizare în batch).

**Complexitatea modelului:**

* Modelul este destul de complex, având în vedere numărul mare de parametri.
* Folosirea dropout-ului poate ajuta la reducerea supradaptării și la îmbunătățirea generalizării modelului.
* Utilizarea normalizării în batch poate ajuta la stabilizarea antrenării și la accelerarea convergenței.

**Potențialul pentru învățarea și generalizarea modelului:**

* Modelul are potențialul de a învăța și generaliza modele complexe de date, dar va necesita o cantitate semnificativă de date de antrenament pentru a face acest lucru eficient.
* Performanța și generalizarea modelului pot fi îmbunătățite prin ajustarea hiperparametrilor și prin utilizarea tehnicilor de regularizare suplimentare.

**Compilare și antrenarea modelului**

model.compile(

    loss='binary\_crossentropy',

    optimizer='adam',

    metrics=['accuracy']

)

* model.compile: Această metodă compilează modelul, specificând funcția de pierdere (loss), optimizatorul (optimizer) și metricile de evaluare (metrics). În acest caz, se utilizează funcția de pierdere 'binary\_crossentropy', optimizatorul 'adam' și metrica 'accuracy' pentru evaluarea performanței modelului.

steps\_per\_epoch = max(1, len(Train\_paths) // BATCH\_SIZE)

validation\_steps = len(Val\_paths) // BATCH\_SIZE + 1

history = model.fit(

    train\_dataset,

    steps\_per\_epoch=steps\_per\_epoch,

    epochs=20,

    validation\_data=val\_dataset,

    validation\_steps=validation\_steps

)

* steps\_per\_epoch: Este calculat numărul de pași pe epocă pentru antrenament. Este setat la maximul dintre 1 și numărul total de imagini de antrenament împărțit la dimensiunea lotului (BATCH\_SIZE). Acest lucru asigură că toate datele de antrenament sunt parcurse complet în fiecare epocă.
* validation\_steps: Este calculat numărul de pași pentru validare. Este setat la numărul total de imagini de validare împărțit la dimensiunea lotului, la care se adaugă 1. Acest lucru se face pentru a asigura că toate datele de validare sunt folosite în timpul evaluării modelului.
* model.fit: Această metodă antrenează modelul pe datele de antrenament. Se furnizează setul de date de antrenament (train\_dataset), numărul de pași pe epocă (steps\_per\_epoch), numărul de epoci (epochs), setul de date de validare (val\_dataset) și numărul de pași pentru validare (validation\_steps). Antrenarea continuă timp de 20 de epoci.

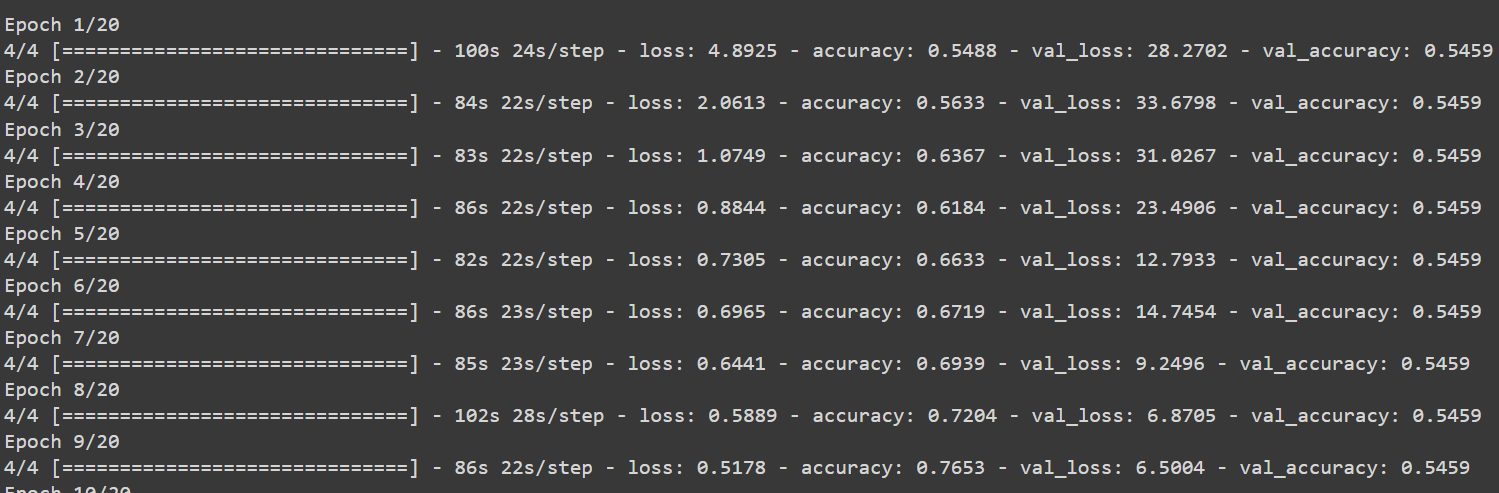


Figura 11 – Rezultatele în urma compilării modelului

Pe măsură ce procesul de antrenare progresează prin mai multe epoci, rețeaua neurală ajustează ponderile în funcție de erorile pe care le face în predicțiile sale. Prin urmare, antrenarea pe mai multe epoci permite modelului să învețe mai multe despre datele de antrenament și să își îmbunătățească performanța în recunoașterea și generalizarea modelelor din datele de antrenament.

**Evaluarea modelului**

loss, acc = model.evaluate(val\_dataset)

print("Testing Acc : ", acc)

print("Testing Loss : ", loss)

Rezultatul evaluării pe setul de testare indică o acuratețe (accuracy) de aproximativ 67.15% și o pierdere (loss) de aproximativ 0.67. Aceste valori reprezintă performanța modelului pe datele de testare, care sunt separate de datele de antrenament și de validare și nu au fost folosite în timpul procesului de antrenare.

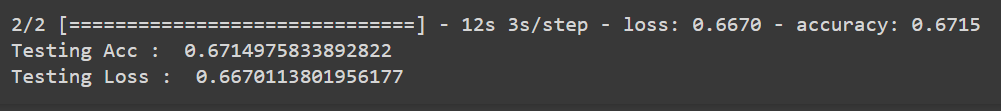


Figura 12 – Evaluarea modelului

**Concluzie**

Analiza detaliată a procesului de creare și antrenare a unui model CNN pentru clasificarea vehiculelor evidențiază eficiența rețelelor neurale artificiale în recunoașterea imaginilor. După importul și pregătirea datelor, modelul a fost definit și antrenat pe setul de date de antrenare. Evaluarea pe setul de testare a indicat o acuratețe de aproximativ 67.15% și o pierdere de aproximativ 0.67, demonstrând potențialul rețelelor neurale în clasificarea precisă a obiectelor din imagini.