

# Editorial – Problema Notăția Bizantină

Soluție realizată de Mihai Nan (mihai.nan@upb.ro)

May 16, 2025

## 1 Descrierea soluției

```
[1]: import warnings
warnings.filterwarnings("ignore", message="Failed to load image Python_
↳extension*")
import pandas as pd
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from torchvision import transforms
from torch.utils.data import DataLoader
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset
from sklearn.metrics import classification_report
import os
import shutil
import cv2
import math
```

- Asigurăm **dezactivarea utilizării plăcii video (GPU)** de către PyTorch, forțând antrenarea modelului doar pe **CPU**.
- Specificăm numele fișierului din care vom prelua datele pentru antrenare (dataset\_train.csv).
- Specificăm dacă vrem sau nu să folosim un set de date pentru validarea modelului de clasificare.

```
[2]: os.environ["CUDA_VISIBLE_DEVICES"] = ""
csv_file = 'dataset_train.csv'
use_eval = True
```

### 1.1 Preprocesarea datelor

Acest fragment de cod realizează preprocesarea datelor pentru un set de imagini etichetate:

- Încarcă datele dintr-un fișier CSV într-un **DataFrame** Pandas.

- Curăță etichetele eliminând spațiile inutile la începutul și sfârșitul fiecărui nume de clasă.
- Creează o mapare de la etichete text la indici numerici (ex: 0, 1, ... 7).
- Înlocuiește șirurile de caractere din coloana **Effect** cu valorile numerice corespunzătoare.
- Dacă variabila `use_eval` este activată:
  - Datele sunt împărțite într-un set de antrenare (90%) și unul de testare (10%), păstrând proporțiile dintre clase.
  - Ambele seturi sunt salvate în fișiere CSV (`train.csv` și `test.csv`).
- Dacă `use_eval` este dezactivat:
  - Întregul set este folosit pentru antrenare și salvat într-un singur fișier CSV (`train.csv`).

Această etapă pregătește datele pentru a fi utilizate ulterior de către un `CustomDataset` care va încărca imaginile și etichetele la cerere.

```
[3]: df = pd.read_csv(csv_file)
df['Effect'] = df['Effect'].str.strip()
label_names = sorted(df['Effect'].unique())
label_to_idx = {label: idx for idx, label in enumerate(label_names)}
idx_to_label = {v: k for k, v in label_to_idx.items()}
df['Effect'] = df['Effect'].map(label_to_idx)

if use_eval:
    df_train, df_test = train_test_split(df, test_size=0.1, random_state=42,
    ↳stratify=df['Effect'])
    df_train.to_csv("train.csv", index=False)
    df_test.to_csv("test.csv", index=False)
else:
    df_train = df
    df_train.to_csv("train.csv", index=False)
```

## 1.2 Încărcarea imaginilor cu etichete în PyTorch

În proiectele de învățare automată care lucrează cu imagini, este important să putem încărca datele într-un mod clar și flexibil. PyTorch oferă o clasă de bază numită `Dataset`, iar noi o putem personaliza pentru a citi imaginile și etichetele din fișierele noastre.

Mai jos este un exemplu de clasă personalizată numită `CustomDataset`, care:

- citește un fișier `.csv` care conține, pentru fiecare imagine, calea ei și eticheta asociată;
- convertește etichetele text în numere (ex: fiecare tip de neumă primește un indice numeric, începând cu 0);
- primește o transformată pe care o putem aplica pentru procesarea imaginilor;
- permite accesul la o anumită imagine folosind un index (`idx`);
- **încarcă imaginea doar atunci când este cerută** – adică **la cerere**, nu toate deodată;
- returnează o pereche formată din imagine și eticheta sa, gata de folosit pentru antrenarea unui model.

```
[4]: class CustomDataset(Dataset):
    def __init__(self, csv_file, transform=None):
        self.data_frame = pd.read_csv(csv_file)
        self.unique_labels = sorted(set(self.data_frame['Effect']))
        self.label_to_idx = {label: idx for idx, label in enumerate(self.
→unique_labels)}
        self.data_frame['Effect'] = self.data_frame['Effect'].map(self.
→label_to_idx)
        self.transform = transform

    def __len__(self):
        return len(self.data_frame)

    def __getitem__(self, idx):
        img_path = self.data_frame.iloc[idx]['Path']
        image = Image.open(img_path).convert('L')
        if self.transform:
            image = self.transform(image)
        effect = self.data_frame.iloc[idx]['Effect']
        try:
            label = int(effect)
        except ValueError:
            label = -1
        return image, label
```

### 1.3 Configurarea seturilor de date și a transformărilor pentru antrenare și testare

- Se stabilește dimensiunea mini-batch-ului (`batch_size = 8`), adică numărul de imagini procesate simultan într-un pas de antrenare.
- Se definesc transformările pentru **antrenare** (`train_transform`), aplicate pe fiecare imagine:
  - **Redimensionare** la  $48 \times 48$  pixeli.
  - **Rotire aleatoare** a imaginii cu un unghi de până la  $\pm 10^\circ$ .
  - **Decupare aleatoare** (cu redimensionare) pentru a introduce variație de scală și proporții.
  - **Ajustări aleatorii de luminozitate și contrast** pentru a simula condiții de iluminare diferite.
  - **Conversie la tensor**, necesară pentru a putea fi procesată de PyTorch.
- Se creează setul de date pentru antrenare (`train_dataset`) folosind `train.csv` și transformările definite.
- Se definește un `DataLoader` pentru antrenare care va oferi batch-uri de imagini amestecate, utile pentru generalizarea modelului.
- Dacă `use_eval` este activat:
  - Se definesc transformări **mai simple pentru testare** (`test_transform`), doar redimen-

sionare și conversie la tensor – fără augmentări, pentru a evalua performanța în condiții controlate.

- Se creează setul de date de testare și un `DataLoader` corespunzător, fără amestecare (shuffle), asigurând evaluarea consistentă.

```
[5]: batch_size = 8
train_transform = transforms.Compose([
    transforms.Resize((48, 48)),
    transforms.RandomRotation(10),
    transforms.RandomResizedCrop(48, scale=(0.9, 1.0), ratio=(0.9, 1.1)),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
])

train_dataset = CustomDataset(csv_file="train.csv", transform=train_transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
if use_eval:
    test_transform = transforms.Compose([
        transforms.Resize((48, 48)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5], std=[0.5])
    ])
    test_dataset = CustomDataset(csv_file="test.csv", transform=test_transform)
    test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

## 1.4 Definirea unui model CNN simplu pentru clasificare de imagini

- Se definește o clasă `BasicCNN`, care moștenește `nn.Module` și reprezintă o rețea neurală convoluțională simplă pentru clasificare.
- **Constructorul `__init__`:**
  - `conv1`: Primul strat convoluțional primește imagini cu un singur canal (alb-negru) și extrage 32 de hărți de caracteristici folosind filtre  $3 \times 3$ , cu păstrarea dimensiunii (`padding=1`).
  - `conv2`: Al doilea strat convoluțional primește cele 32 de hărți și produce 64 de hărți noi, tot cu filtre  $3 \times 3$  și `padding`.
  - `pool`: Stratul de max-pooling reduce dimensiunea hărților la jumătate (dimensiune spațială redusă cu `kernel=2` și `stride=2`).
  - `dropout`: Regularizare cu dropout (25%) pentru a reduce overfitting-ul.
  - `fc1`: Primul strat complet conectat (fully connected) primește vectorul *aplatizat* (64 hărți  $\times 24 \times 24$  după pooling) și produce 128 de unități ascunse.
  - `fc2`: Stratul final care mapează caracteristicile extrase la numărul de clase (ieșire pentru clasificare).
- **Metoda `forward`:**
  - Aplică ReLU după fiecare strat convoluțional.
  - Aplică pooling după al doilea strat convoluțional.

- Aplică dropout pentru regularizare.
- Aplatizează (flatten) rezultatul pentru a fi introdus în straturile complet conectate.
- Aplică ReLU după primul strat complet conectat.
- Returnează rezultatul stratului final `fc2`, care conține scorurile brute pentru fiecare clasă (logits).

```
[6]: import torch.nn.functional as F

class BasicCNN(torch.nn.Module):
    def __init__(self, num_classes):
        super(BasicCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3,
        ↪padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
        ↪padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout = nn.Dropout(0.25)
        self.fc1 = nn.Linear(64 * 24 * 24, 128)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.dropout(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

## 1.5 Inițializarea modelului, a funcției de pierdere și a optimizatorului

- `device = "cpu"`: Specifică faptul că modelul și datele vor fi rulate pe CPU. Dacă se dorește folosirea GPU-ului, se poate înlocui cu `device = "cuda"` (dacă este disponibil).
- `epochs = 10` stabilește câte ori întregul set de date de antrenament va fi parcurs complet în timpul procesului de învățare.
- `model = BasicCNN(num_classes=len(label_to_idx))`: Creează o instanță a modelului `BasicCNN`, configurat să producă câte un scor pentru fiecare dintre clasele posibile (determinate din `label_to_idx`).
- `model.to(device)`: Mută modelul pe dispozitivul specificat (`cpu` în acest caz), pentru a asigura că toate calculele se fac acolo (inclusiv în timpul antrenării).
- `criterion = nn.CrossEntropyLoss()`: Definește funcția de pierdere care va fi folosită pentru antrenare. `CrossEntropyLoss` este potrivită pentru probleme de clasificare multi-clasă, combinând softmax și negative log-likelihood.
- `optimizer = optim.Adam(model.parameters(), lr=0.001)`: Inițializează algoritmul de optimizare Adam, care actualizează parametrii modelului în timpul antrenării. Se setează o rată de învățare (`lr`) de 0.001, o valoare standard de pornire.

```
[7]: device = "cpu"
epochs = 50
model = BasicCNN(num_classes=len(label_to_idx))
model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

## 1.6 Bucla de antrenare a modelului

- Se parcurg toate epocile (epochs) pentru antrenarea modelului.
- La începutul fiecărei epoci se setează modelul în modul de antrenare (`model.train()`), ceea ce activează straturi precum Dropout și BatchNorm (dacă există).
- `total_loss` este folosit pentru a aduna pierderea de-a lungul mini-batch-urilor.

Pentru fiecare batch:

- Se mută imaginile (inputs) și etichetele (labels) pe dispozitivul specificat (cpu sau cuda).
- `optimizer.zero_grad()` resetează gradientii acumulați din pasul anterior.
- Se obține ieșirea modelului pentru batch-ul curent: `outputs = model(inputs)`.
- Se calculează eroarea în raport cu etichetele reale: `loss = criterion(outputs, labels)`.
- Se face backpropagation: `loss.backward()`.
- Se actualizează parametrii modelului: `optimizer.step()`.

La finalul fiecărei epoci, se afișează pierderea totală pentru epoca respectivă, pentru monitorizarea progresului.

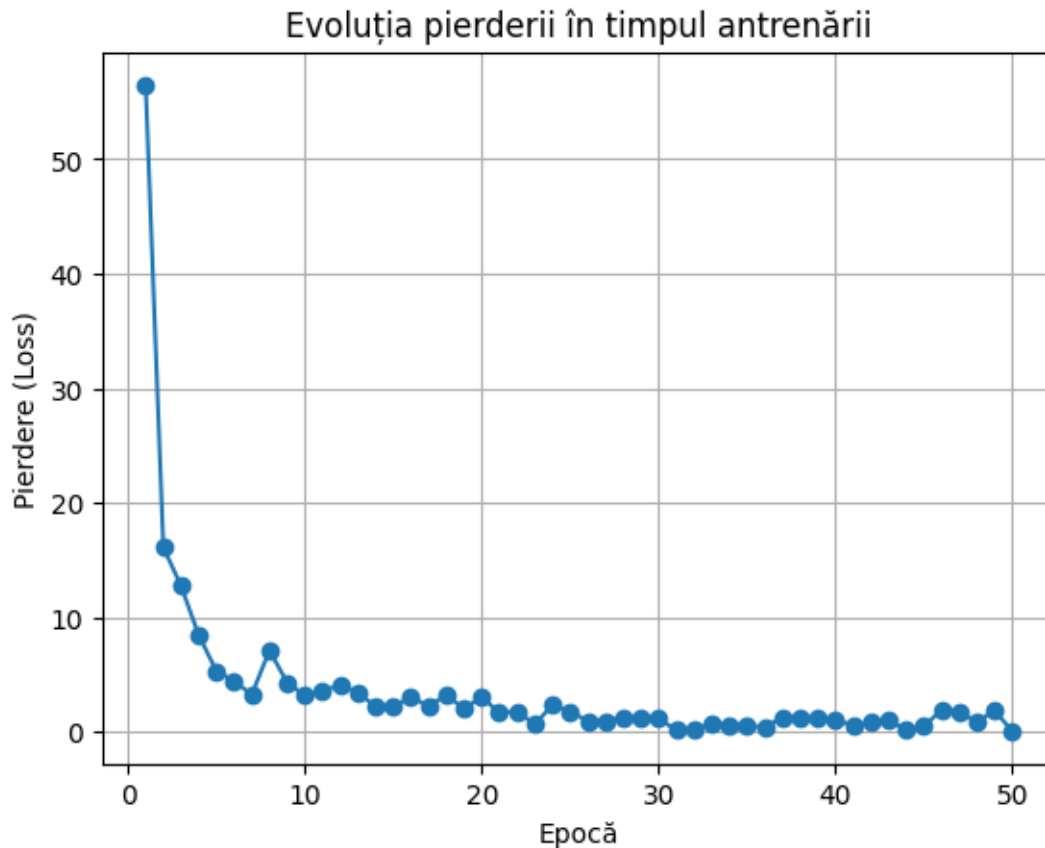
```
[8]: losses = []
for epoch in range(epochs):
    model.train()
    total_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    losses.append(total_loss)
    print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss:.4f}")

plt.plot(range(1, epochs + 1), losses, marker='o')
plt.xlabel('Epocă')
plt.ylabel('Pierdere (Loss)')
plt.title('Evoluția pierderii în timpul antrenării')
plt.grid(True)
plt.show()
```

Epoch 1/50, Loss: 56.4607  
Epoch 2/50, Loss: 16.1084  
Epoch 3/50, Loss: 12.7374  
Epoch 4/50, Loss: 8.4428  
Epoch 5/50, Loss: 5.3097  
Epoch 6/50, Loss: 4.4313  
Epoch 7/50, Loss: 3.2590  
Epoch 8/50, Loss: 7.1221  
Epoch 9/50, Loss: 4.3036  
Epoch 10/50, Loss: 3.1765  
Epoch 11/50, Loss: 3.6340  
Epoch 12/50, Loss: 4.0401  
Epoch 13/50, Loss: 3.4452  
Epoch 14/50, Loss: 2.1956  
Epoch 15/50, Loss: 2.1550  
Epoch 16/50, Loss: 3.0662  
Epoch 17/50, Loss: 2.2291  
Epoch 18/50, Loss: 3.2304  
Epoch 19/50, Loss: 2.0018  
Epoch 20/50, Loss: 3.0602  
Epoch 21/50, Loss: 1.6550  
Epoch 22/50, Loss: 1.8029  
Epoch 23/50, Loss: 0.6459  
Epoch 24/50, Loss: 2.3566  
Epoch 25/50, Loss: 1.7637  
Epoch 26/50, Loss: 0.9588  
Epoch 27/50, Loss: 0.9424  
Epoch 28/50, Loss: 1.3006  
Epoch 29/50, Loss: 1.1701  
Epoch 30/50, Loss: 1.1685  
Epoch 31/50, Loss: 0.1601  
Epoch 32/50, Loss: 0.1895  
Epoch 33/50, Loss: 0.7575  
Epoch 34/50, Loss: 0.6300  
Epoch 35/50, Loss: 0.5534  
Epoch 36/50, Loss: 0.3827  
Epoch 37/50, Loss: 1.2743  
Epoch 38/50, Loss: 1.2440  
Epoch 39/50, Loss: 1.2196  
Epoch 40/50, Loss: 1.0543  
Epoch 41/50, Loss: 0.5883  
Epoch 42/50, Loss: 0.8446  
Epoch 43/50, Loss: 1.0881  
Epoch 44/50, Loss: 0.1409  
Epoch 45/50, Loss: 0.5022  
Epoch 46/50, Loss: 1.8893  
Epoch 47/50, Loss: 1.8038  
Epoch 48/50, Loss: 0.8274

Epoch 49/50, Loss: 1.8845

Epoch 50/50, Loss: 0.0592



## 1.7 Evaluarea modelului pe setul de test

- `if use_eval`: verifică dacă evaluarea modelului pe datele de test este activată.
- `model.eval()` setează modelul în modul de evaluare, dezactivând comportamente specifice antrenării (ex: Dropout).
- Se inițializează listele `y_true` și `y_pred` pentru a stoca etichetele reale și predicțiile făcute de model.
- Blocul `with torch.no_grad()`: dezactivează calculul gradientului, economisind memorie și accelerând inferența.
- Pentru fiecare batch din `test_loader`:
  - Se mută imaginile pe dispozitivul folosit (`cpu` sau `cuda`).
  - Se calculează ieșirile modelului (`outputs`).
  - Se obțin predicțiile finale prin alegerea clasei cu scorul cel mai mare (`torch.argmax`).
  - Predicțiile și etichetele reale sunt convertite în liste Python și adăugate la `y_pred` și `y_true`.
- La final, se afișează un raport de clasificare (`classification_report`) care arată măsuri precum precizia, recall și F1-score pentru fiecare clasă, folosind denumirile claselor (`label_names`).



- Argumentul `zero_division=0` previne erorile în calcul atunci când o clasă nu are predicții.

```
[9]: if use_eval:
    model.eval()
    y_true, y_pred = [], []
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs = inputs.to(device)
            outputs = model(inputs)
            preds = torch.argmax(outputs, dim=1).cpu().numpy()
            y_pred.extend(preds)
            y_true.extend(labels.numpy())
    print(classification_report(y_true, y_pred, target_names=label_names,
↪zero_division=0))
```

	precision	recall	f1-score	support
-1	1.00	1.00	1.00	5
-2	1.00	1.00	1.00	5
-4	1.00	0.67	0.80	3
0	0.83	1.00	0.91	5
1	1.00	1.00	1.00	9
4	1.00	1.00	1.00	5
A	1.00	1.00	1.00	5
B	1.00	1.00	1.00	5
accuracy			0.98	42
macro avg	0.98	0.96	0.96	42
weighted avg	0.98	0.98	0.97	42

## 1.8 Salvarea modelului și a etichetelor

- `torch.save()` salvează un dicționar care conține:
  - `model_state_dict`: starea curentă a parametrilor modelului (greutăți și bias-uri).
  - `label_to_idx`: dicționarul care mapează numele claselor la indici, esențial pentru interpretarea predicțiilor.
- Fișierul '`model_nn.pth`' este folosit pentru a păstra aceste informații pe disc.
- Această salvare permite ulterior încărcarea modelului cu aceleași *greutăți* și folosirea corectă a etichetelor fără a pierde corespondența între indici și clase.

```
[10]: torch.save({
    'model_state_dict': model.state_dict(),
    'label_to_idx': label_to_idx
}, 'model_nn.pth')
```

## 1.9 Pregătirea mediului și încărcarea modelului pentru evaluare

- Se definește directorul `patches_dir` pentru a stoca fragmentele de imagini procesate.
- Dacă directorul există deja, este șters complet (`shutil.rmtree`) pentru a începe cu un spațiu curat.
- Se creează directorul `patches_dir` nou.
- Se încarcă punctul de control salvat anterior (`model_nn.pth`) folosind `torch.load`.
- Se extrag din checkpoint parametrii modelului (`model_state_dict`) și dicționarul de etichete (`label_to_idx`).
- Se creează o instanță a modelului `BasicCNN` cu numărul corect de clase (8).
- Se încarcă starea parametrilor în model (`load_state_dict`) și se setează modelul în modul evaluare (`eval()`).
- Se recrează un codificator de etichete (`LabelEncoder`), folosind clasele sortate pentru a putea decodifica predicțiile modelului în denumiri de clase.
- Se încarcă fișierul `ground_truth.csv` într-un `DataFrame` `pandas`, ce conține etichetele reale pentru evaluare.

```
[11]: patches_dir = "patches"

if os.path.exists(patches_dir):
    shutil.rmtree(patches_dir)
os.makedirs(patches_dir)

checkpoint = torch.load('model_nn.pth')
model_state_dict = checkpoint['model_state_dict']
label_to_idx = checkpoint['label_to_idx']

model = BasicCNN(num_classes=8)
model.load_state_dict(model_state_dict)
model.eval()

label_encoder = LabelEncoder()
label_encoder.classes_ = np.array([key for key, _ in sorted(label_to_idx.items())]) # Recreate the encoder

eval_df = pd.read_csv("ground_truth.csv")
```

## 1.10 Funcția `encode_prediction`

- Această funcție primește o listă de predicții (de tip șir de caractere) și le transformă într-un format numeric codificat, cel cerut în enunțul problemei.
- Se pornește cu o listă goală `result` în care se vor adăuga valorile codificate.
- Pentru primul element din predicții:
  - Dacă este 'A' sau 'B', se adaugă 0 în listă (acestea nu afectează tonul, neavând valoare vocalică).
  - Altfel, se adaugă valoarea numerică corespunzătoare predicției (convertită din șir în întreg).
- Pentru elementele următoare:

- Dacă predicția este 'A' sau 'B', se repetă ultima valoare din listă (nu se schimbă tonul).
- Altfel, se adaugă suma ultimei valori din listă și valoarea numerică a predicției curente.
- La final, lista `result` este convertită într-un singur șir, unde elementele sunt separate prin caracterul pipe |.

```
[12]: def encode_prediction(predictions):
    result = []
    for pred in predictions:
        if len(result) == 0 and (pred == 'A' or pred == 'B'):
            result.append(0)
        elif len(result) == 0:
            result.append(int(pred))
        elif pred == 'A' or pred == 'B':
            result.append(result[-1])
        else:
            result.append(result[-1] + int(pred))
    final_result = str(result[0])
    for item in result[1:]:
        final_result += "|" + str(item)
    return final_result
```

### 1.11 Funcția `remove_background`

- Scopul funcției este să elimine fundalul unei imagini și să păstreze doar partea relevantă (obiectul principal).
- Dacă imaginea are 3 canale (este color), se convertește în grayscale (alb-negru).
- Se aplică o binarizare simplă (`threshold`) pentru a separa părțile deschise (fundalul) de cele închise.
- Se aplică un filtru median (`medianBlur`) pentru a reduce zgomotul.
- Se aplică o binarizare adaptivă pentru a obține un contur mai clar al obiectului.
- Imaginea binarizată este inversată (`bitwise_not`) pentru a identifica componentele conectate ale obiectului.
- Se identifică toate componentele conectate din imagine și se analizează dimensiunea fiecăreia.
- Se păstrează doar componentele cu aria mai mare sau egală cu `min_area`, eliminând zgomotele mici.
- Se găsesc coordonatele pixelilor valizi și se determină un dreptunghi minim care încapsulează obiectul (bounding box).
- Se decupează imaginea originală conform acestui dreptunghi pentru a obține doar obiectul fără fundal.
- Dacă nu se găsesc componente valide, se returnează imaginea binarizată inițială.

```
[13]: def remove_background(img, blur_ksize=3, min_area=30):
    if len(img.shape) == 3:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    else:
        img = img.copy()
    _, img = cv2.threshold(img, 160, 255, cv2.THRESH_BINARY)
    blurred = cv2.medianBlur(img, blur_ksize)
    thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.
    →THRESH_BINARY, 15, 10)
    inverted = cv2.bitwise_not(thresh)
    num_labels, labels, stats, _ = cv2.connectedComponentsWithStats(inverted,
    →connectivity=8)
    clean = np.zeros_like(inverted)
    for i in range(1, num_labels):
        if stats[i, cv2.CC_STAT_AREA] >= min_area:
            clean[labels == i] = 255
    coords = cv2.findNonZero(clean)
    if coords is None:
        return img
    x, y, w, h = cv2.boundingRect(coords)
    cropped = img[y:y+h, x:x+w]
    return cropped
```

```
[14]: img = cv2.imread('eval_img/40.png')
cropped_img = remove_background(img)
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].imshow(img, cmap='gray')
axs[0].set_title("Imagine originală")
axs[0].axis('off')

axs[1].imshow(cropped_img, cmap='gray')
axs[1].set_title("Imagine fără fundal")
axs[1].axis('off')

plt.show()
```



## 1.12 Identificarea zonelor care conține neume

La antrenare lucrăm cu imagini simple, fiecare conținând o singură notă muzicală (neumă). Însă, la evaluare putem primi imagini compuse, care conțin mai multe neume grupate. Pentru a le putea clasifica corect, mai întâi trebuie să identificăm și să extragem din aceste imagini compuse zonele

(patch-urile) unde apar neumele individuale. Funcția `extract_patches` ne ajută să facem exact acest lucru, separând fiecare neumă într-un patch separat pentru clasificare ulterioară.

- **Eliminarea fundalului:** Se apelează funcția `remove_background` pentru a păstra doar regiunile importante din imagine.
- **Binarizare adaptivă:** Se aplică un prag adaptiv invers pentru a evidenția obiectele (contururile) în imagine.
- **Detectarea contururilor:** Se identifică toate contururile externe (obiectele distincte).
- **Filtrarea după dimensiune:** Se ignoră contururile a căror arie (lățime \* înălțime) este mai mică decât `min_area` pentru a elimina zgomotul.
- **Extragerea patch-urilor:** Pentru fiecare contur valid, se extrage o regiune dreptunghiulară din imagine, cu o mică margine (padding) pentru a include context suplimentar.
- **Sortarea patch-urilor:** Patch-urile sunt sortate pe orizontală, după poziția lor pe axa x, pentru a păstra ordinea spațială din imagine.

```
[15]: def extract_patches(img, min_area=30, pad=2):
    img = remove_background(img)
    thresh = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.
    ↪THRESH_BINARY_INV, 15, 10)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.
    ↪CHAIN_APPROX_SIMPLE)
    patches = []
    for cnt in contours:
        x, y, w, h = cv2.boundingRect(cnt)
        if w * h < min_area:
            continue
        x_start = max(x - pad, 0)
        y_start = max(y - pad, 0)
        x_end = min(x + w + pad, img.shape[1])
        y_end = min(y + h + pad, img.shape[0])
        patch = img[y_start:y_end, x_start:x_end]
        patches.append((x_start, patch))
    patches = [patch for _, patch in sorted(patches, key=lambda p: p[0])]
    return patches
```

```
[16]: img = cv2.imread('eval_img/41.png')
patches = extract_patches(img)
cols = 4
rows = math.ceil((len(patches) + 1) / cols)

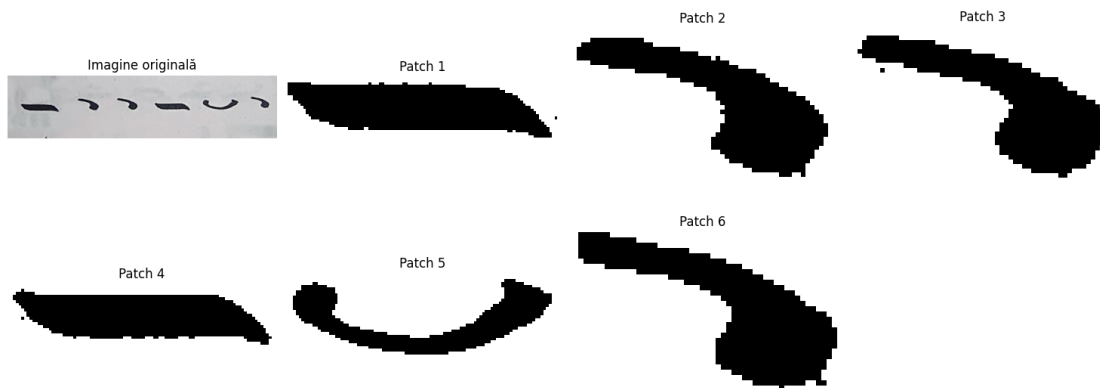
plt.figure(figsize=(15, 3 * rows))

plt.subplot(rows, cols, 1)
plt.title('Imagine originală')
plt.axis('off')
plt.imshow(img, cmap='gray')

for i, patch in enumerate(patches):
```

```
plt.subplot(rows, cols, i + 2)
plt.title(f'Patch {i + 1}')
plt.axis('off')
plt.imshow(patch, cmap='gray')

plt.tight_layout()
plt.show()
```



Funcția `make_square` este folosită pentru a transforma imagini dreptunghiulare în imagini pătrate, ceea ce este important deoarece imaginile folosite pentru antrenarea rețelei neurale convoluționale (CNN) au avut dimensiuni pătrate și uniforme.

În procesul nostru, imaginile folosite la antrenare sunt toate pătrate (de exemplu 48x48 pixeli), însă în timpul prelucrării imaginilor compuse sau a patch-urilor extrase, este posibil să avem zone dreptunghiulare de dimensiuni diferite. Această funcție adaugă margini albe în jurul imaginii astfel încât să devină pătrată, iar apoi redimensionează rezultatul la dimensiunea dorită.

Astfel, `make_square` asigură compatibilitatea imaginilor extrase cu modelul CNN, evitând deformările care apar la redimensionarea directă a imaginilor dreptunghiulare la dimensiuni pătrate. Acest pas ajută la păstrarea proporțiilor obiectelor din imagini și contribuie la o mai bună performanță a modelului.

```
[17]: def make_square(image, target_size=48):
    h, w = image.shape
    size = max(h, w)
    square = np.full((size, size), 255, dtype=image.dtype)
    y_offset = (size - h) // 2
    x_offset = (size - w) // 2
    square[y_offset:y_offset+h, x_offset:x_offset+w] = image
    resized = cv2.resize(square, (target_size, target_size))
    return resized
```

```
[18]: img = cv2.imread('eval_img/41.png', cv2.IMREAD_GRAYSCALE)
patches = extract_patches(img)
```

```

cols = 2
rows = len(patches)

plt.figure(figsize=(8, 3 * rows))

for i, patch in enumerate(patches):
    # Original patch
    plt.subplot(rows, cols, 2*i + 1)
    plt.title(f'Patch {i + 1} original')
    plt.axis('off')
    plt.imshow(patch, cmap='gray')

    # Squared patch
    squared_patch = make_square(patch)
    plt.subplot(rows, cols, 2*i + 2)
    plt.title(f'Patch {i + 1} pătrat')
    plt.axis('off')
    plt.imshow(squared_patch, cmap='gray')

plt.tight_layout()
plt.show()

```

Patch 1 pătrat



Patch 1 original



Patch 2 pătrat



Patch 2 original



Patch 3 pătrat



Patch 3 original



Patch 4 pătrat



Patch 4 original



Patch 5 pătrat



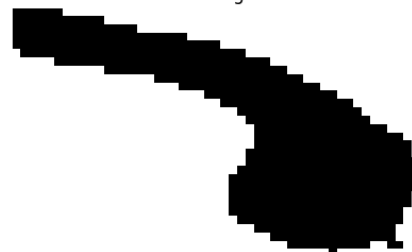
Patch 5 original



Patch 6 pătrat



Patch 6 original





- Funcția `predict_image` primește calea către o imagine, un model de clasificare și un encoder de etichete.
- Imaginea este citită în tonuri de gri; dacă nu există imaginea, funcția returnează o listă goală.
- Se extrag patch-urile din imagine folosind `extract_patches`. Dacă nu se găsesc patch-uri, se returnează listă goală.
- Pentru fiecare patch valid (cu dimensiuni minime), se creează o versiune pătrată și redimensionată la 48x48 pixeli.
- Patch-urile procesate sunt salvate în folderul “patches” cu un nume bazat pe numele fișierului original și un index.
- Fiecare patch este normalizat și transformat într-un vector unidimensional pentru a fi pregătit pentru model.
- Toate patch-urile procesate sunt convertite într-un tensor și transmise modelului pentru predicție.
- Rezultatele modelului sunt convertite în etichete originale folosind encoderul, iar lista de etichete returnată.
- Funcția permite clasificarea mai multor obiecte (patch-uri) dintr-o singură imagine compusă.

```
[19]: def predict_image(filepath, model, label_encoder):
    img = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)
    if img is None:
        return []

    patches = extract_patches(img)
    if not patches:
        return []

    processed = []
    idx = 1
    path = filepath
    for patch in patches:
        if patch.shape[0] < 5 or patch.shape[1] < 5:
            continue
        resized = make_square(patch)
        sample_id = path.split("/")[1]
        sample_id = sample_id.split(".")[0]
        cv2.imwrite("patches/" + sample_id + "_" + str(idx) + ".png", resized)
        idx += 1
        flat = resized.astype('float32') / 255.0
        processed.append(flat.flatten())

    if not processed:
        return []

    X_eval = np.array(processed)
    X_eval = X_eval.reshape(-1, 1, 48, 48)
```

```
X_eval_tensor = torch.tensor(X_eval).float()
y_pred = model(X_eval_tensor).argmax(dim=1).numpy()
labels = label_encoder.inverse_transform(y_pred)
return labels.tolist()
```

```
[20]: prediction = predict_image("eval_img/41.png", model, label_encoder)
print(prediction)
```

```
['1', '-1', '-1', '1', '0', '-1']
```

### 1.13 Generarea fișierului pentru submitie

- Inițializează o listă **results** pentru a stoca rezultatele evaluării.
- Parcurge fiecare rând din **eval\_df**, care conține informații despre imaginile de evaluare.
- Pentru fiecare **datapointID** (calea către o imagine), verifică dacă fișierul există; dacă nu, adaugă un rezultat gol.
- Dacă fișierul există, apelează **predict\_image** pentru a obține etichetele prezise de model pentru patch-urile imaginii.
- Preia eticheta țintă (răspunsul corect) din coloana "**answer**" a DataFrame-ului.
- Codifică predicțiile folosind funcția **encode\_prediction** pentru a obține o reprezentare comparabilă.
- Adaugă în lista **results** un tuple cu: calea imaginii, etichetele prezise, eticheta țintă și codificarea predicțiilor.
- Acest proces permite evaluarea modelului pe un set de imagini și compararea rezultatelor cu valorile așteptate.

```
[21]: results = []
for _, row in eval_df.iterrows():
    datapoint = row["datapointID"]
    if not os.path.exists(datapoint):
        results.append((datapoint, []))
        continue
    predicted_labels = predict_image(datapoint, model, label_encoder)
    target = row["answer"]
    result = encode_prediction(predicted_labels)
    results.append((datapoint, predicted_labels, target, result))
```

```
[22]: submission = eval_df.copy()
answer = []
for datapoint, _, target, result in results:
    if target != result:
        print(f"{datapoint} -> {target} vs {result}")
    answer.append(result)
submission["answer"] = answer
submission.to_csv("submission.csv", index=False)
```

```
eval_img/40.png -> 0|1|0|1|2|3|2|1|0|-1|0 vs
0|4|0|0|-1|3|-1|0|1|1|-3|-4|-8|-9|-10|-11|-10
```

```

eval_img/41.png -> 1|0|-1|0|1|0 vs 1|0|-1|0|0|-1
eval_img/51.jpg -> 0|1|-1|0|0|0|1 vs 0|1|-1|0|1|1|2
eval_img/52.jpg -> 1|1|-3|-7|-6|-8 vs 1|1|-3|-3|-2|-4
eval_img/61.png -> 1|2|1|2|1|0 vs 1|1|0|1|0|-1
eval_img/75.png -> 0|0|1|0|-1 vs 0|0|1|5|4
eval_img/76.png -> -4|-5|-6 vs 0|-1|-2
eval_img/1062.png -> 1|0|0|0|-2|-4|0|-2|-6|-6 vs 1|0|4|4|2|0|4|2|-2|-2
eval_img/1069.png -> 1|1|-3|-5|-6|-6|-8|-9|-9|-8|-7 vs
1|1|-3|-5|-6|-7|-9|-10|-10|-9|-8
eval_img/1072.png -> -1|-5|-5|-5|-5|-6|-6|-8|-8|-8 vs
-1|-5|-5|-5|-5|-6|-7|-9|-5|-5
eval_img/1073.png -> 0|0|-2|-2|-2|-2|-1|-2|-1|3 vs 4|4|2|6|6|6|7|6|7|11
eval_img/1081.png -> -2|2|1|2|6|6|6|6|2|2|2 vs -2|2|1|2|6|10|10|10|6|6|6
eval_img/1088.png -> 1|-3|-3|-3|-2|-4|-4|-6|-6|-5 vs 1|-3|1|1|2|0|0|-2|-2|-1
eval_img/1099.png -> 0|0|-4|-3|-7|-11|-11|-13|-15 vs 4|4|0|1|-3|-7|-7|-9|-11
eval_img/2025.png -> -2|-2|-2|2|1 vs -2|-2|-2|-1|3|2

```

[ ]: