

CT Scan Classification

Documentație

Proiect Machine Learning








Hermeneanu Mara

Grupa 312

Introducere

Scopul proiectului a fost acela de a antrena modele de machine learning pe un set de date constând în tomografii ale unor plamani și a clasifica imaginile CT în una din cele trei categorii: nativ (0), arterial (1), venos (2). În paginile ce urmează voi descrie abordarea folosită și tipurile de modele implementate (MLP și CNN).

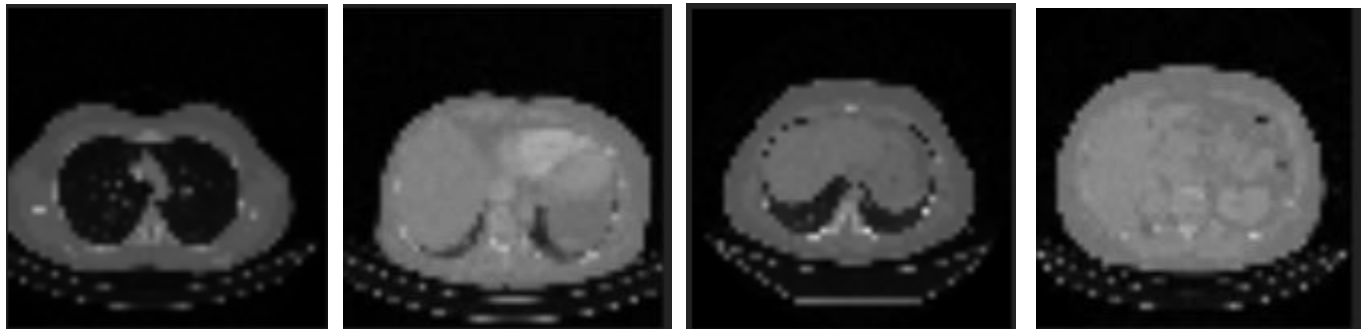
Seturile de date

- ▶  test
- ▶  train
- ▶  validation
-  sample_submission.txt
-  test.txt
-  train.txt
-  validation.txt

Fișierele train.txt și validation.txt conțin pe fiecare linie numele imaginii și label-ul respectiv, în formatul: nume_imagine, label_imagine. Imaginile propriu-zise se regasesc în folderele train și validation.

Fișierul test.txt conține pe fiecare linie denumirea unei imagini, iar imaginile propriu-zise se găsesc în folderul test.

Imaginile sunt în format png, grayscale, cu o rezoluție de 50x50 pixeli.



(cateva din CT Scan-urile de train afisate in program)

1. Multi Layer Perceptron Classifier

Încărcarea seturilor de date

Funcția **read_from_txt(filename, foldername)** citește din fișier numele imaginilor și label-urile corespunzătoare și le adaugă în două liste separate. Având numele imaginilor și numele folderului în care se află imaginile, folosesc în continuare **Image.open()** pentru a le încărca într-un np.array, iar lista de label-uri o transform de asemenea într-un np.array. Funcția va returna np.array-ul de imagini și np.array-ul de labels.

```
def read_from_txt(filename, foldername):

    imglist = []
    lablist = []

    with open(filename, "r") as file:
        lines = [line.strip() for line in file if line.strip()] #ignora liniile goale

    for line in lines:
        imglab = line.strip().split(',') #liniile sunt de forma nume-image,label-image
        imglist.append(imglab[0]) #lista cu numele imaginilor
        lablist.append(int(imglab[1])) #lista cu labeluri

    #afisarea catorva imagini
    # imagini = [Image.open('./'+ foldername+'/'+ +imgname) for imgname in imglist[:5]]
    # for img in imagini:
    #     img.show()

    #np array cu imaginile
    x = np.array([np.array(Image.open('./'+ foldername+'/'+ +imgname)) for imgname in imglist])
    #print(x.shape)

    #np array cu labelurile
    y = np.array(lablist)

    return x,y
```

Normalizarea datelor

Pentru normalizarea datelor am folosit o funcție implementată la laborator **normalize_data(train_data, test_data)**, care primește ca parametri datele de antrenare, respectiv de testare și folosește **preprocessing.StandardScaler()** pentru a le normaliza. Înainte de a normaliza imaginile a fost nevoie sa fac reshape pentru a reduce o dimensiune. Astfel, pentru imaginile de train am schimbat shape-ul din (15000,50,50) în (15000,2500), pentru imaginile de validation din (4500,50,50) în (4500,2500), iar pentru imaginile de test din (3900,50,50) în (3900,2500).

```
def normalize_data(train, test=None): #functie folosita pentru a normaliza datele
    scaler = preprocessing.StandardScaler()
    scaler.fit(train)
    scaler_training_data = scaler.transform(train)

    if test is not None:
        scaler_test_data = scaler.transform(test)
        return scaler_training_data, scaler_test_data

    return scaler_training_data
```

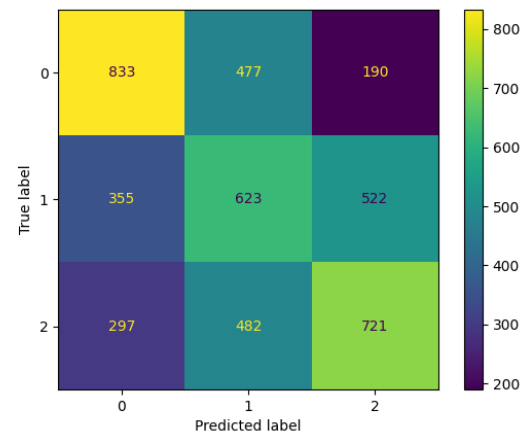
Antrenarea modelului și hyperparameters

Pentru alegerea parametrilor am încercat mai multe combinații posibile, cea mai reușită fiind cea cu funcția de activare „relu”, solver-ul implicit „adam”, learning_rate „adaptive”, max_iter = 2000 și hidden_layer_sizes=(200,100,50). Acuratețea maximă obținută a fost în jur de 0.52.

Comparație între solvelele *adam* și *sgd*

```
Acuratetea pe datele de train: 0.9948
Acuratetea pe datele de validare: 0.5075555555555555
[[833 477 190]
 [355 623 522]
 [297 482 721]]
```

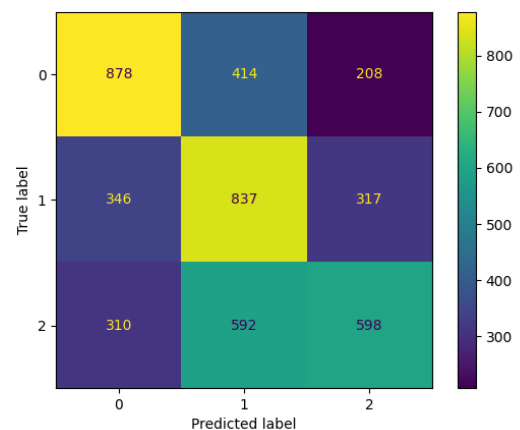
	precision	recall	f1-score	support
0	0.56	0.56	0.56	1500
1	0.39	0.42	0.40	1500
2	0.50	0.48	0.49	1500
accuracy			0.48	4500
macro avg	0.49	0.48	0.48	4500
weighted avg	0.49	0.48	0.48	4500



(rezultate obținute cu adam, max_iter=1000, learning_rate='adaptive', activation = 'relu')

```
Acuratetea pe datele de train: 1.0
Acuratetea pe datele de validare: 0.48733333333333334
[[878 414 208]
 [346 837 317]
 [310 592 598]]
```

	precision	recall	f1-score	support
0	0.57	0.59	0.58	1500
1	0.45	0.56	0.50	1500
2	0.53	0.40	0.46	1500
accuracy			0.51	4500
macro avg	0.52	0.51	0.51	4500
weighted avg	0.52	0.51	0.51	4500



(rezultate obținute cu sgd, max_iter=1000, learning_rate='adaptive' activation = 'relu')

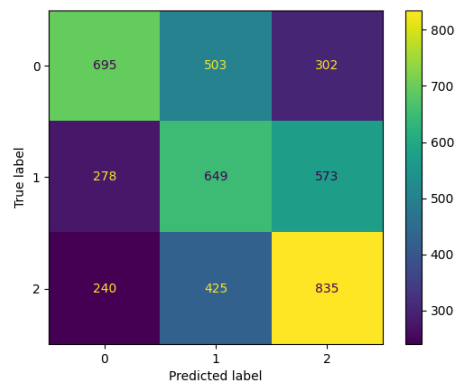
Comparație între learning rate *constant* și *adaptive/relu* și *logistic*

```

Acuratetea pe datele de train: 0.9974666666666666
Acuratetea pe datele de validare: 0.4886666666666667
[[695 503 302]
 [278 649 573]
 [240 425 835]]

```

	precision	recall	f1-score	support
0	0.57	0.46	0.51	1500
1	0.41	0.43	0.42	1500
2	0.49	0.56	0.52	1500
accuracy			0.48	4500
macro avg	0.49	0.48	0.48	4500
weighted avg	0.49	0.48	0.48	4500



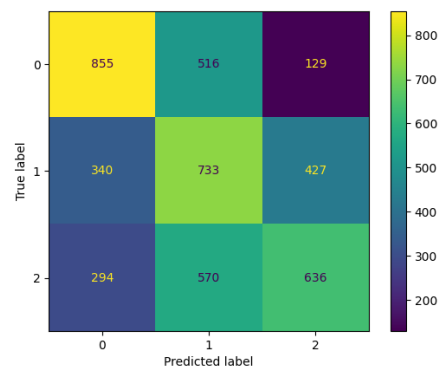
(rezultate obținute cu learning rate constant, max_iter=2000, activation = 'relu', solver='adam')

```

Acuratetea pe datele de train: 0.9971333333333333
Acuratetea pe datele de validare: 0.5188888888888888
[[855 516 129]
 [340 733 427]
 [294 570 636]]

```

	precision	recall	f1-score	support
0	0.57	0.57	0.57	1500
1	0.40	0.49	0.44	1500
2	0.53	0.42	0.47	1500
accuracy			0.49	4500
macro avg	0.50	0.49	0.50	4500
weighted avg	0.50	0.49	0.50	4500



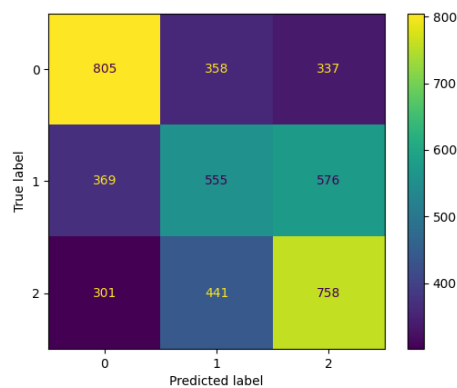
(rezultate obținute cu learning rate adaptive, max_iter=2000, activation = 'relu', solver='adam')

```

Acuratetea pe datele de train: 0.9954666666666667
Acuratetea pe datele de validare: 0.4671111111111111
[[805 358 337]
 [369 555 576]
 [301 441 758]]

```

	precision	recall	f1-score	support
0	0.55	0.54	0.54	1500
1	0.41	0.37	0.39	1500
2	0.45	0.51	0.48	1500
accuracy			0.47	4500
macro avg	0.47	0.47	0.47	4500
weighted avg	0.47	0.47	0.47	4500



(rezultate obținute cu learning rate adaptive, max_iter=2000, activation = 'logistic', solver='adam')

2. Convolutional Neural Network

Având în vedere că acuratețea obținută cu MLP a fost una destul de nesatisfăcătoare, am decis să folosesc un algoritm bazat pe CNN, acest tip de rețea fiind extrem de populară pentru clasificare de imagini. Pentru a implementa acest tip de clasificare am folosit biblioteca PyTorch, ghidându-mă după documentația oficială.

Încărcarea seturilor de date

Pentru a face batch training am implementat o clasa CTScanData care moștenește clasa Dataset și conține metodele `__getitem__` și `__len__` necesare pentru construirea de DataLoaders pornind de la obiectele de tip CTScanData. În constructorul clasei CTScanData încarc imaginile și label-urile corespunzătoare într-un mod similar cu funcția `read_from_txt(filename, foldername)`, cu deosebirea că transform lista de np.array într-o listă de tensori (`torch_transf.ToTensor()`) pe care îi normalizez (`torch_transf.Normalize(media,dev)`).

Arhitectura rețelei

Primul model a fost conceput având 3 straturi convolutionale, 3 straturi de dropout și 3 straturi liniare cu 4320,2160 și în final 3 features. Pentru straturile convoluționale am ales să dublez numărul de out_channels pe fiecare strat, iar pentru kernel am ales o dimensiune constantă de 3x3. Straturile de dropout anulează elementele tensorului de input cu o anumită probabilitate, împiedicând totodată overfitting-ul. Pentru aceste probabilități am ales valorile 0.5, 0.4 și 0.3.

```

class CNN(torch.nn.Module):

    def __init__(self):
        super(CNN, self).__init__() #(W-F+2P)/S + 1
        self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 60, kernel_size = 3) #kernel = filter size = 3x3 stride
        self.conv1_drop = torch.nn.Dropout2d(p=0.5) #probability
        self.conv2 = torch.nn.Conv2d(in_channels = 60, out_channels = 120, kernel_size = 3)
        self.conv2_drop = torch.nn.Dropout2d(p=0.4) #impiedica overfitting-ul
        self.conv3 = torch.nn.Conv2d(in_channels = 120, out_channels = 240, kernel_size = 3) #padding ca imaginea sa nu deva
        self.conv3_drop = torch.nn.Dropout2d(p=0.3)

        self.fc1 = torch.nn.Linear(in_features=240*9*9, out_features=4320)
        self.fc2 = torch.nn.Linear(in_features=4320, out_features=2160)
        self.fc3 = torch.nn.Linear(in_features=2160, out_features=3) #3 clase

    def forward(self, x):
        x = torch.nn.functional.relu(torch.nn.functional.max_pool2d(self.conv1_drop(self.conv1(x)), 2))
        x = torch.nn.functional.relu(torch.nn.functional.max_pool2d(self.conv2_drop(self.conv2(x)), 2)) #max pooling pe 2x2
        x = torch.nn.functional.relu(self.conv3_drop(self.conv3(x)))

        x = torch.flatten(x, 1) #aplatizeaza toate dimensiunile mai putin dimensiunea batch-ului

        x = torch.nn.functional.relu(self.fc1(x))
        x = torch.nn.functional.relu(self.fc2(x))
        x = self.fc3(x)

        return x

#initializare retea CNN
network = CNN()
optimizer = torch.optim.Adam(network.parameters(), lr=0.0005, amsgrad=True) #0.0007/9

```

În metoda forward am ales să folosesc max pooling pentru feature extraction cu un filtru de dimensiune 2x2 după fiecare strat convoluțional, astfel că dimensiunile imaginii vor fi înjumătățite. Funcția de activare folosită a fost relu. Optimizer-ul folosit a fost adam cu un learning rate de 0.0005, iar acuratețea obținută cu acest model a fost în jur de 0.70.

Încercând să obțin o acuratețe mai mare decât cea a primului model, am adăugat un strat convoluțional în plus și am adăugat straturile de dropout doar după straturile convoluționale 2 și 3. De asemenea, am mărit succesiv dimensiunea kernel-ului, am micșorat numărul de features pentru straturile liniare și am folosit ca ultimă funcție activatoare funcția softmax cu dim=1. În ceea ce privește optimizer-ul, am ales să rămână adam, însă am mărit learning rate-ul la 0.0009. Acuratețea obținută cu acest model a fost mai bună, oscilând între 0.72 și 0.75.

```

def __init__(self):
    super(CNN, self).__init__() #(W-F+2P)/S + 1
    self.conv1 = torch.nn.Conv2d(in_channels = 1, out_channels = 25, kernel_size = 2) #kernel = filter size = 3
    self.conv2 = torch.nn.Conv2d(in_channels = 25, out_channels = 50, kernel_size = 3)
    self.conv2_drop = torch.nn.Dropout2d(p=0.5) #impiedica overfitting-ul
    self.conv3 = torch.nn.Conv2d(in_channels = 50, out_channels = 100, kernel_size = 3) #padding ca imaginea sa
    self.conv3_drop = torch.nn.Dropout2d(p=0.4)
    self.conv4 = torch.nn.Conv2d(in_channels = 100, out_channels = 200, kernel_size = 5)

    self.fc1 = torch.nn.Linear(in_features=200*5*5,out_features= 2000)
    self.fc2 = torch.nn.Linear(in_features=2000, out_features=1000)
    self.fc3 = torch.nn.Linear(in_features=1000,out_features=3) #3 clase

def forward(self, x):

    x = torch.nn.functional.relu(torch.nn.functional.max_pool2d(self.conv1(x),2))
    x = torch.nn.functional.relu(torch.nn.functional.max_pool2d(self.conv2_drop(self.conv2(x)), 2)) #max pooling

    x = torch.nn.functional.relu(self.conv3_drop(self.conv3(x)))
    x = torch.nn.functional.relu(self.conv4(x))

    x = torch.flatten(x,1) #aplatizeaza toate dimensiunile mai putin dimensiunea batch-ului

    x = torch.nn.functional.relu(self.fc1(x))
    x = torch.nn.functional.relu(self.fc2(x))
    x = self.fc3(x)

    return torch.nn.functional.log_softmax(x,dim=1)

#initializare retea CNN
network = CNN()
optimizer = torch.optim.Adam(network.parameters(), lr=0.0009,amsgrad=True) #0.0007/9

```

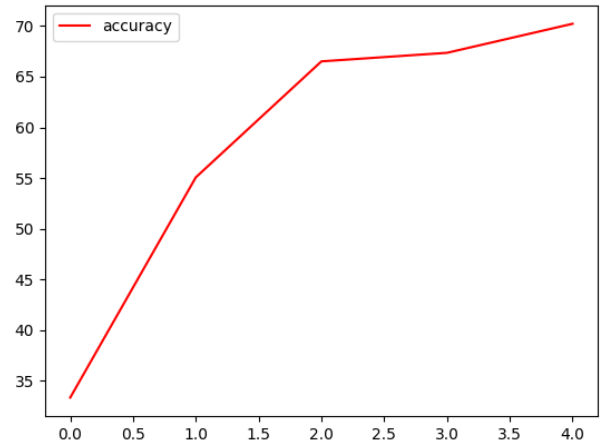
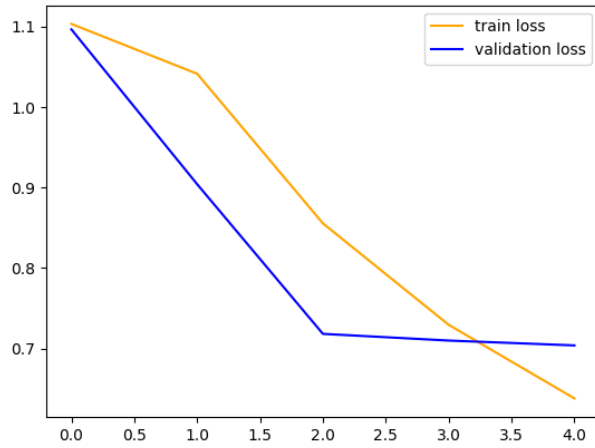
Pentru fiecare epocă am calculat `training_loss`, respectiv `validation_loss` pentru a vedea cum evoluează modelul (dacă există semne de underfitting sau overfitting) și am reținut numărul de imagini corect clasificate pentru a calcula acuratețea după formula $100 * \text{imagini_corect_clasificate} / \text{total_imagini}$.

Rezultate pentru primul model

```

epoch: 0 no. of correct predictions: 1500 train loss: 1.1035332322120666 validation loss: 1.0967475626203749 accuracy: 33.333333333333336 %
epoch: 1 no. of correct predictions: 2478 train loss: 1.041563003063202 validation loss: 0.9042059659957886 accuracy: 55.06666666666667 %
epoch: 2 no. of correct predictions: 2993 train loss: 0.8555370263258616 validation loss: 0.7181452684932285 accuracy: 66.51111111111111 %
epoch: 3 no. of correct predictions: 3031 train loss: 0.7294004527727763 validation loss: 0.7098194903797573 accuracy: 67.35555555555555 %
epoch: 4 no. of correct predictions: 3160 train loss: 0.6378693256775538 validation loss: 0.7037924117512173 accuracy: 70.22222222222223 %

```

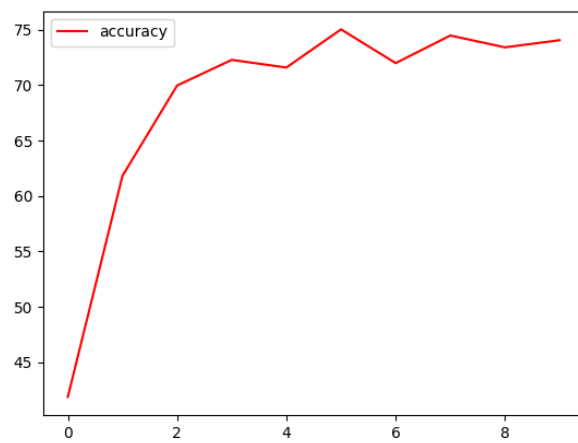
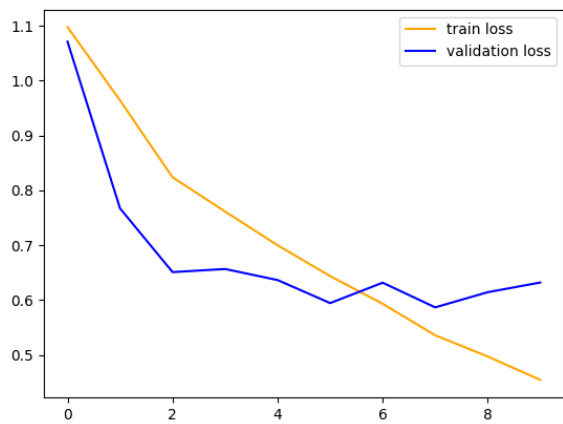
(graficele pentru 5 epoci)

Rezultate pentru cel de-al doilea model

```

epoch: 0 no. of correct predictions: 1630 train loss: 1.094764187335968 validation loss: 1.0842547621045793 accuracy: 36.22222222222222 %
epoch: 1 no. of correct predictions: 2967 train loss: 0.9477256405353546 validation loss: 0.7324407236916678 accuracy: 65.93333333333334 %
epoch: 2 no. of correct predictions: 2995 train loss: 0.8097537058591843 validation loss: 0.7057769588061742 accuracy: 66.55555555555556 %
epoch: 3 no. of correct predictions: 3123 train loss: 0.7366199016571044 validation loss: 0.6723786098616463 accuracy: 69.4 %
epoch: 4 no. of correct predictions: 3146 train loss: 0.6792328721284866 validation loss: 0.6745028410639081 accuracy: 69.91111111111111 %
epoch: 5 no. of correct predictions: 3251 train loss: 0.618203172981739 validation loss: 0.7264936123575483 accuracy: 72.24444444444444 %
epoch: 6 no. of correct predictions: 3291 train loss: 0.5558222118020058 validation loss: 0.7103581922394889 accuracy: 73.13333333333334 %
epoch: 7 no. of correct predictions: 3295 train loss: 0.4943298524618149 validation loss: 0.7338602202279227 accuracy: 73.22222222222223 %
epoch: 8 no. of correct predictions: 3302 train loss: 0.4567221403121948 validation loss: 0.7322242021560669 accuracy: 73.37777777777778 %
epoch: 9 no. of correct predictions: 3408 train loss: 0.4066237771511078 validation loss: 0.7950591240610395 accuracy: 75.73333333333333 %
epoch: 10 no. of correct predictions: 3357 train loss: 0.37146999448537826 validation loss: 0.8710760372025627 accuracy: 74.6 %
epoch: 11 no. of correct predictions: 3325 train loss: 0.33540573209524155 validation loss: 1.0511067645890373 accuracy: 73.88888888888889 %
epoch: 12 no. of correct predictions: 3302 train loss: 0.3029607509076595 validation loss: 1.0361601829528808 accuracy: 73.37777777777778 %
epoch: 13 no. of correct predictions: 3320 train loss: 0.28225578367710114 validation loss: 1.0801085114479065 accuracy: 73.77777777777777 %
epoch: 14 no. of correct predictions: 3322 train loss: 0.2590579880774021 validation loss: 1.1124598741531373 accuracy: 73.82222222222222 %
epoch: 15 no. of correct predictions: 3335 train loss: 0.24330526873469352 validation loss: 1.260667223589761 accuracy: 74.11111111111111 %
epoch: 16 no. of correct predictions: 3393 train loss: 0.2084288489818573 validation loss: 1.2951015165873936 accuracy: 75.4 %
epoch: 17 no. of correct predictions: 3295 train loss: 0.21405306361615659 validation loss: 1.3064557535307748 accuracy: 73.22222222222223 %
epoch: 18 no. of correct predictions: 3359 train loss: 0.18038619510829448 validation loss: 1.5172708817890712 accuracy: 74.64444444444445 %
epoch: 19 no. of correct predictions: 3383 train loss: 0.17123679023236035 validation loss: 1.3901131067957198 accuracy: 75.17777777777778 %
epoch: 20 no. of correct predictions: 3306 train loss: 0.16814850240945817 validation loss: 1.554792308807373 accuracy: 73.46666666666667 %
epoch: 21 no. of correct predictions: 3362 train loss: 0.156465988041057586 validation loss: 1.5504847645759583 accuracy: 74.71111111111111 %
epoch: 22 no. of correct predictions: 3388 train loss: 0.13531308736652137 validation loss: 1.5759953669139317 accuracy: 75.28888888888889 %
epoch: 23 no. of correct predictions: 3410 train loss: 0.12747122306376696 validation loss: 1.5117425441741943 accuracy: 75.77777777777777 %
epoch: 24 no. of correct predictions: 3334 train loss: 0.12619945239275693 validation loss: 1.5890131711959838 accuracy: 74.08888888888889 %

```



(graficele pentru 10 epoci)