Tema 2 Invatare Automata

Student Pietraru Alfred Andrei
Grupa 343C2

1 MLP pe atribute extrase din etapa 1

In etapa 1, atributele au fost extrase folosind 4 algoritmi.

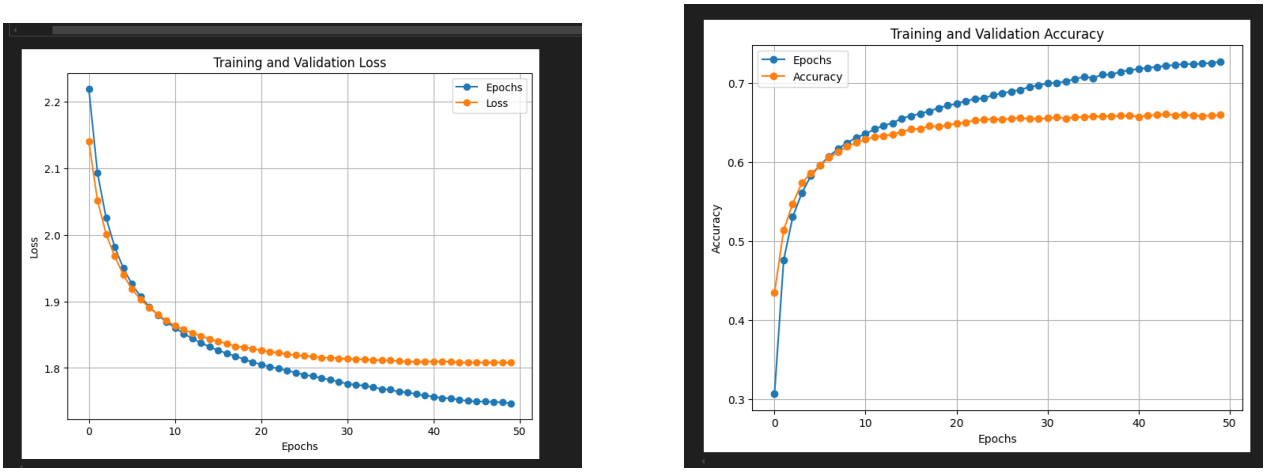
1. ORB - folosit pentru a extrage punctele cheie din imagine, unde diferenta de intensitati dintre pixeli era mai mare decat un anumit **threshold**, sugerand ca acolo ar putea fi frontiera dintre 2 obiecte.
2. HoG - Histogram of Oriented Gradients - calculeaza directia in care variaza intensitatea luminoasa a pixelilor in regiunea cheie gasita, pentru a evidentia linia de separare intre cele 2 obiecte.
3. Kmeans - atat ORB cat si HoG sunt aplicati pe fiecare imagine in parte generand un numar foarte mare de feature-uri, Kmeans este folosit pentru a alege doar cele mai reprezentative K feature-uri, de exemplu in procesul de antrenare pentru un **batch de date de 200 de imagini**, aplic ORB + HoG si obtin cateva mii de feature-uri, aplicand Kmeans, selectez doar 10 dintre acestea cele mai reprezentative pentru batch-ul respectiv, in final obtin **vocabularul cu dimensiunea (1500, 32)**
4. RandomForest - calculez pentru fiecare imagine in parte o histograma care arata care dintre feature din vocabular se regasesc in imaginea curenta, fiecare histograma este ulterior folosita ca input, alaturi de labelul asociat ca si input pentru modelul de RandomForest care selecteaza care feature-uri sunt cele mai relevante pentru task-ul de clasificare. Fiind foarte multe feature-uri, repet de 2 ori acest proces, ajungand de la 1500 de feature-uri la 64 in cazul FashionMNIST si de la **COMPLETEAZA AICI PENTRU FRUITS** la 128.

La o rulare initiala am observat ca aproximativ o treime din imagini nu au absolut niciun punct cheie in urma rularii ORB. Pentru a remedia problema am scazut parametrul threshold de la 36 la 25, scazand astfel numarul imaginilor fara puncte cheie la doar cateva mii. In situatia in care imaginea in continuare nu are niciun punct cheie, aplic direct pe ea algoritmul HoG. Dar pentru a face acest lucru, dimensiunea initiala de (28, 28) nu ma prea ajuta. Asa ca fac resize la (64, 64). Modificand dimensiunea, obtin mai multi descriptori. In cazul in care imaginea este corect, o aduc in format grayScale pentru a putea aplica acesti algoritmi.

- functia de pierdere - CrossEntropyLoss
- optimizator - torch.Adam, cu lr initial de 1e-5, si weight_decay_factor de 1e-4
- antrenarea am facut-o pe 50 de epoci.
- pentru a varia learning rate-ul in timpul antrenarii, folosesc un learning rate scheduler, ReduceLROnPlateau, cu patience de 3, modific lr pentru a se putea potrivi mai bine pentru datele de validare.
- ca functii de activare dupa fiecare pereche de strat linear si strat de batch normalization, aplic functia ReLU, iar pentru ultimul strat aplic Softmax, pentru a aduce rezultatele in intervalul [0, 1] reprezentand probabilitatea cu care modelul crede ca imaginea face parte dintr-o anumita clasa.
- pentru a rezolva problema overfiting-ului dupa fiecare strat liniar folosesc BatchNormalization1d ca metoda de regularizare si o strategie de early stopping, cu patience de 7, sunt memorati parametrii pentru modelul cu acuratetea cea mai mare si este aplicat pe datele de test.
- acuratetea este 64%, f1 score este 64%, recall este 64%, precision este 64% si a durat 02:56
- Arhitectura modelului pentru FashionMNIST (in stanga se afla tipul de strat folosit, in dreapta numarul de neuroni care alcatuiesc stratul respectiv. Modelul are 348746 parametrii.
    1. (fc1): Linear(in=64, out=128, bias=True) - 8320
    2. (bn1): BatchNorm1d(128, eps=1e-05, momentum=0.1) - 256
    3. (fc2): Linear(in=128, out=256, bias=True) - 33024

4. (bn2): BatchNorm1d(256, eps=1e-05, momentum=0.1) - 512
5. (fc3): Linear(in=256, out=512, bias=True) - 131584
6. (bn3): BatchNorm1d(512, eps=1e-05, momentum=0.1) - 1024
7. (fc4): Linear(in=512, out=256, bias=True) - 131328
8. (bn4): BatchNorm1d(256, eps=1e-05, momentum=0.1) - 512
9. (fc5): Linear(in=256, out=128, bias=True) - 32896
10. (bn5): BatchNorm1d(128, eps=1e-05, momentum=0.1) - 256
11. (fc6): Linear(in=128, out=64, bias=True) - 8256
12. (bn6): BatchNorm1d(64, eps=1e-05, momentum=0.1) - 128
13. (fc7): Linear(in=64, out=10, bias=True) - 650

1.1 Rezultate FashionMNIST



(a) Loss (b) Accuracy

Figure 1: Graficele de loss si de acuratete pentru arhitectura de MLP pe feature-uri

Learning rate-ul a fost ales in mod corespunzator, graficele pentru pierdere si acuratete evolueaza impreuna intr-o maniera controlata, fara a creea modificari majore de la o epoca la alta. Problema pe care o intampin este ca indiferent cum modific arhitectura, si oricum as incerca sa adaug parametrii, acuratetea modelului nu creste, sugerand o greseala in modul in care sunt extrase feature-urile. Observam ca dupa epoca 30 acuratetea modelului nu se mai imbunatateste, iar loss-ul ramane relativ stabil, aces lucru fiind cauzat poate si de faptul ca learning rate scheduler-ul, reduce valoarea learning rate-ului cu cateva ordine de marime, facand procesul de invatare extrem de lent.

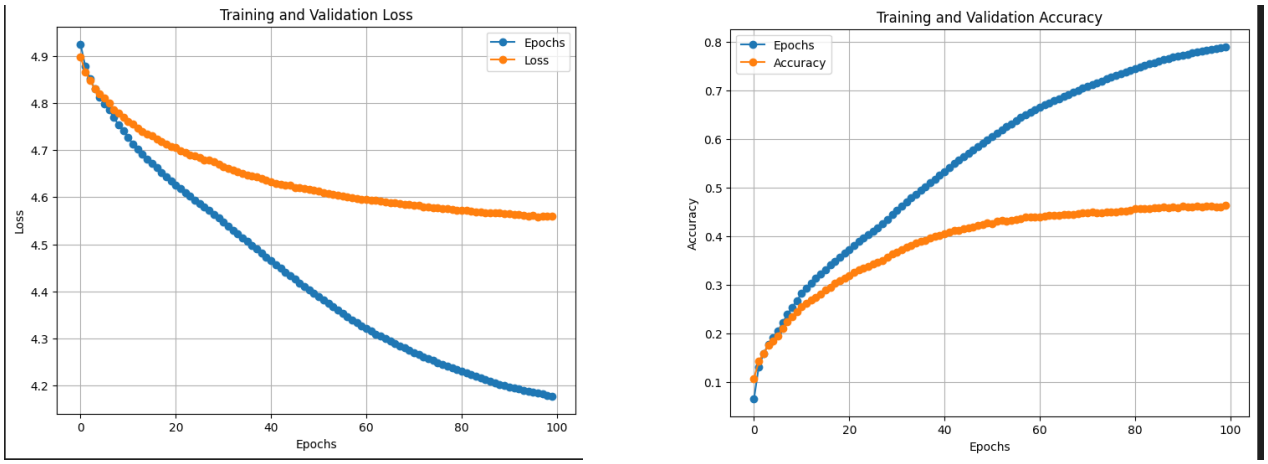
1.2 Rezultate Fruits

Modelul are 153235 de parametrii care pot fi invatati si are urmatoarea arhitectura:

1. (fc1): Linear(in_features=128, out_features=384, bias=True) 49536
2. (bn1): BatchNorm1d(num_features=384, eps=1e-05, momentum=0.1) 768
3. (fc2): Linear(in_features=384, out_features=1152, bias=True) 443520
4. (bn2): BatchNorm1d(num_features=1152, eps=1e-05, momentum=0.1) 2304
5. (fc3): Linear(in_features=1152, out_features=3456, bias=True) 3984768
6. (bn3): BatchNorm1d(num_features=3456, eps=1e-05, momentum=0.1) 6912
7. (fc4): Linear(in_features=3456, out_features=1152, bias=True) 3982464
8. (fc5): Linear(in_features=1152, out_features=384, bias=True) 442752

9. (bn5): BatchNorm1d(num_features=384, eps=1e-05, momentum=0.1) 768
10. (fc6): Linear(in_features=384, out_features=128, bias=True) 49280
11. (bn6): BatchNorm1d(num_features=128, eps=1e-05, momentum=0.1) 256
12. (fc7): Linear(in_features=128, out_features=141, bias=True) 18189

- batch size de 200, numarul de epoci este de 100, care au fost executate toate, modelul avand o continua imbunatatire pe tot parcursul antrenarii, fiind ca si date de intrare histogramele obtinute din extragerea de feature-uri, singura preprocesare aplicata este normalizarea acestora in intervalul [0, 1]
- functia de eroare este CrossEntropyLoss
- optimizatorul folosit este torch.Adam, cu learning rate de 1e-5, am observat o antrenare mult mai stabila cu learning rate asa de mic, weight_decay_factor de 1e-4,
- folosesc un learning scheduler, ReduceLROnPlateau, cu patience de 3, si reducere a ratei de invatare la 0.3 din valoarea initiala, adica lr = lr * 0.3.
- functiile de activare folosite sunt ReLU care sunt aplicate dupa fiecare strat Liniar si Softmax in layer-ul de clasificare
- antrenarea a durat 7 minute, dar rezultatele sunt slabe, acuratetea este 29% 'f1': 16%, 'recall': 17%, 'precision': 18%,



(a) Loss

(b) Acuratete

Figure 2: Graficele de loss si de acuratete pentru dataset Fruits folosind MLP pe feature-urile extrase in etapa 1

2 Arhitecturi MLP direct pe imagini

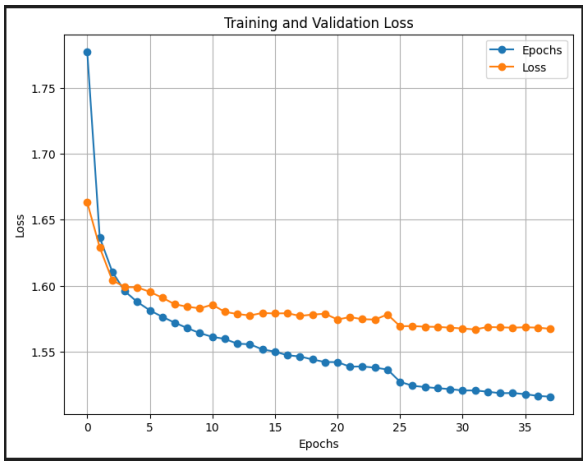
2.1 Preprocesarea Imaginilor

Transformarile aplicate imaginilor sunt toTensor(), pentru a le aduce in range-ul de [0, 1], Resize la (28, 28) pentru a trece inputul rapid prin model, si Normalize(mean=0.5, std=0.5), deoarce am observat experimental ca obtin rezultate mai bune in medie cu 3-4 procente. Aceste transformari sunt aplicate atat pentru FashionMNIST cat si pentru Fruits. Atat pe datele de antrenare cat si pe cele de testare si validare.

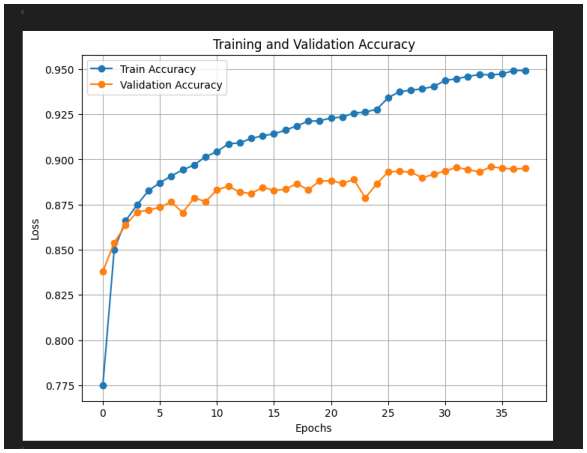
2.2 Rezultate FashionMNIST

- batch size are valoarea 100
- am setat initial 50 de epoci pentru antrenare, dar modelul incepe sa faca overfiting dupa primele 30 de epoci, early stopping a intervenit la epoca 37
- ca metode de regularizare folosesc early stopping cu patience de 7 si BatchNormalization1d dupa fiecare strat liniar

- functia de eroare este CrossEntropyLoss
- optimizatorul folosit este torch.Adam, cu learning rate de 1e-4, weight_decay_factor de 1e-4,
- folosesc un learning scheduler, ReduceLROnPlateau, cu patience de 3, si reducere a ratei de invatare la 0.2 din valoarea initiala, adica lr = lr * 0.2.
- functiile de activare folosite sunt ReLU care sunt aplicate dupa fiecare strat Liniar si Softmax in layer-ul de clasificare
- acuratete: 88%, f1 score: 88%, recall: 88%, precizie : 88%, antrenarea a durat 04:11
- Modelul are 153235 de parametrii care pot fi invatati si are urmatoarea arhitectura:
  1. (fc1): Linear(in=784, out=150, bias=True) - 117750
  2. (bn1): BatchNorm1d(150, eps=1e-05, momentum=0.1) - 300
  3. (fc2): Linear(in=150, out=150, bias=True) - 22650
  4. (bn2): BatchNorm1d(150, eps=1e-05, momentum=0.1) - 300
  5. (fc3): Linear(in=150, out=75, bias=True) - 11325
  6. (bn3): BatchNorm1d(75, eps=1e-05, momentum=0.1) - 150
  7. (fc4): Linear(in=75, out=10, bias=True) - 760



(a) Loss



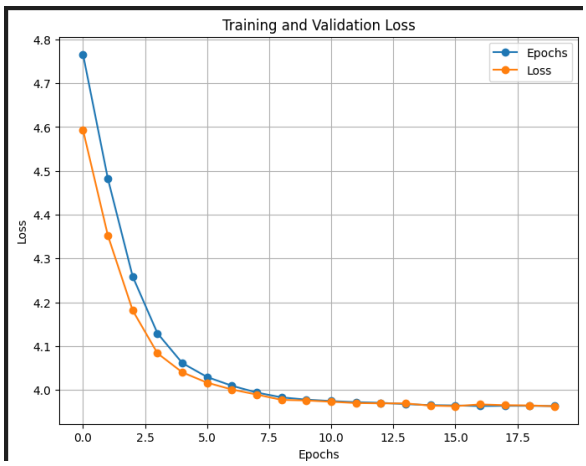
(b) Acuratete

Figure 3: Graficele de loss si de acuratete folosind MLP direct pe imagini

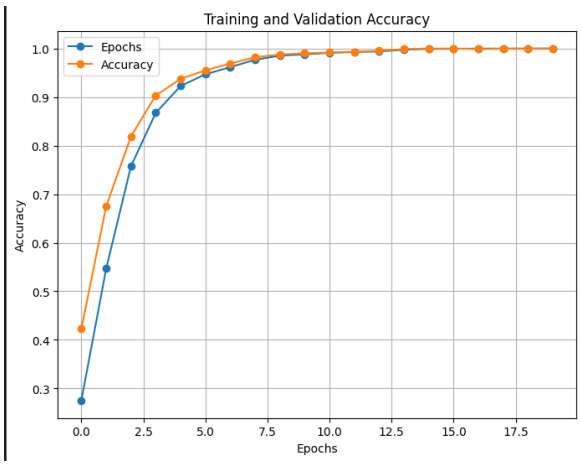
2.3 Rezultate Fruits

Am folosit aceleasi transformari atat pe training, cat si pe validation si test. Acestea sunt ToTensor(), Resize((28, 28)) si Normalize(0.5, 0.5)

- batch size este de 100 si am antrenat pentru 20 de epoci.
- ca functie de pierdere folosesc CrossEntropyLoss, iar ca optimizator folosesc Adam cu learning rate de 1e-4, si weight_decay_factor de 1e-4
- folosesc un learning rate scheduler, cu patience de 3,
- ca tehnici de regularizare folosesc early stopping cu patience de 6, acuratetea maxima o obtinusem la epoca a 14 -a, chiar si in situatia in care aveam mai mult de 20 de epoci, era foarte posibil modelul se opreasca deoarce nu apareau modificari la acuratetea pe datele de validare, BatchNormalization2d
- ca functii de activare folosesc ReLU dupa fiecare strat liniar in afara de ultimul, iar dupa ultimul strat liniar folosesc Softmax pentru a obtine probabilitatea ca imaginea sa faca parte dintr-o clasa, pentru fiecare clasa in parte.
- acuratetea: 95%, f1 score: 92%, recall: 92%, precizie: 94%, rularea a durat 6 minute si 43 de secunde



(a) Loss



(b) Acuratete

Figure 4: Graficele de loss si de acuratete folosind MLP direct pe imagini pe dataset-ul Fruits

3 Arhitectura de tip convoluțional

3.1 Preprocesarea Imaginilor

In antrenarea retelei, am folosit tehnici de augmentare a imaginilor din training set, urmatoarele:

1. ToTensor(),
2. v2.RandomHorizontalFlip(),
3. v2.RandomRotation(degrees=90),
4. v2.RandomCrop(size=((28, 28), padding=(4, 4), padding_mode="reflect"),
5. Resize((28, 28)),
6. Normalize(0.5, 0.5)

Pentru setul de testare si de validare am folosit doar ToTensor(), Resize((28, 28)) si Normalizare(0.5, 0.5).

3.2 Rezultate FashionMNIST

- batch size de 100, setat initial pentru o antrenare de 50 de epoci, la epoca 31 se obtine acuratetea maxima, iar la epoca 40 este oprit modelul de catre algoritmul de early stopping.
- ca functie de pierdere folosesc CrossEntropyLoss, iar ca optimizator folosesc Adam cu learning rate de 1e-4, si weight_decay_factor de 1e-4
- folosesc un learning rate scheduler, cu patience de 4,
- ca tehnici de regularizare folosesc early stopping cu patience de 9, BatchNormalization2d si Dropout in stratul liniar de clasificare cu probabilitatea p = 0.3
- parametrii channels = 1, base_depth = 16, expansion = 4 si nr_layers_block = 2, acestia sunt folositi pentru a modifica eficient arhitecturii si pentru a creste complexitatea acesteia
- cu transformari: acuratete: 0.8194, f1 : 0.81, recall : 0.81, precizie: 0.82, timp: 24 min
- fara transformari: acuratete: 0.77 f1 : 0.78, recall: 0.77, precizie: 0.80, timp: 17 min
- Pentru a realiza arhitectura folosesc o structura numit ConvBlock, daca ConvBlock(bottleneck_channels, block_channels, norm_layer) unde norm_layer este BatchNorm2d, structura acesteia este:

```python
class ConvBlock(nn.Module):
    def __init__(self, bottleneck_channels, block_channels, norm_layer):
        super(ConvBlock, self).__init__()
        self.conv1 = conv1x1(block_channels, bottleneck_channels)
        self.norm1 = norm_layer(bottleneck_channels)
        self.conv2 = conv3x3(bottleneck_channels, bottleneck_channels)
        self.norm2 = norm_layer(bottleneck_channels)
        self.conv3 = conv1x1(bottleneck_channels, block_channels)
        self.norm3 = norm_layer(block_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        identity = x
        x = self.norm1(self.relu(self.conv1(x)))
        x = self.norm2(self.relu(self.conv2(x)))
        x = self.norm3(self.relu(self.conv3(x)))
        x += identity
        return self.relu(x)
```

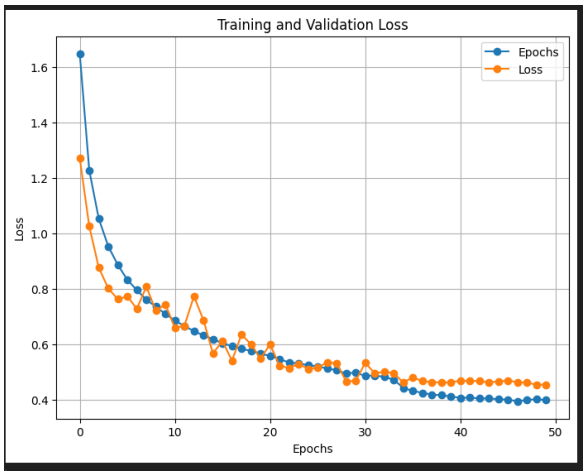
1. (conv1): block_channels $\times$ bottleneck_channels $\times$ 1
2. (norm1): bottleneck_channels $\times$ 2
3. (conv2): bottleneck_channels $\times$ bottleneck_channels $\times$ 9
4. (norm2): bottleneck_channels $\times$ 2
5. (conv3): block_channels $\times$ bottleneck_channels $\times$ 1
6. (norm3): block_channels $\times$ 2

Pentru a realiza arhitectura retele, folosesc ConvBlock, acesta este diferit prin faptul ca se foloseste de input-ul de la un moment anterior de timp in straturile viitoare, pastrand din feature-urile care s-ar putea pierde in urma executarii operatiilor de convolutie. Ca functii de activare, folosesc ReLU si GlobalAveragePooling. Parametrii sunt identici atat in situatia in care se aplica augmentari pe imaginile din setul de antrenare cat si atunci cand sunt folosite doar imaginile initiale, singura transformare aplicata fiind ToTensor().

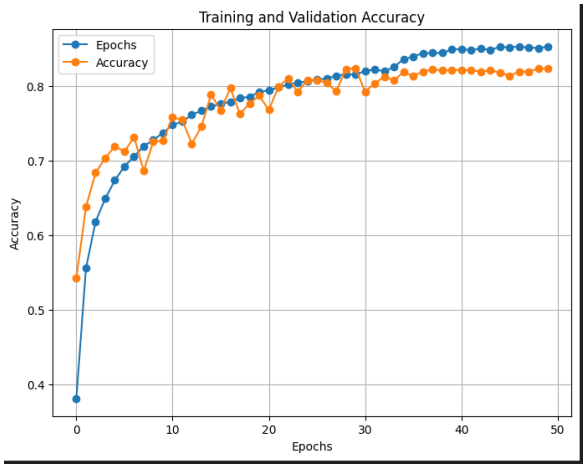
```python
class DeepConvNet(nn.Module):
    def __init__(self, channels, base_depth, expansion, nr_layer_blocks, num_classes=10):
        super(DeepConvNet, self).__init__()
        self.base_depth = base_depth
        self.conv1 = nn.Conv2d(channels, self.base_depth, kernel_size=7, stride=2, padding=3,bias=False)
        self.relu = nn.ReLU(inplace=True)
        depth1 = self.base_depth * expansion
        self.conv_depth1 = conv1x1(self.base_depth, depth1, stride=1)
        self.norm_depth1 = nn.BatchNorm2d(depth1)
        self.layer1 = nn.Sequential(
            *[ConvBlock(self.base_depth, depth1, nn.BatchNorm2d)] * nr_layer_blocks
        )
        depth2 = depth1 * expansion
        self.conv_depth2 = conv1x1(depth1, depth2, 2)
        self.norm_depth2 = nn.BatchNorm2d(depth2)
        self.layer2 = nn.Sequential(
            *[ConvBlock(depth1, depth2, nn.BatchNorm2d)] * nr_layer_blocks
        )
        depth3 = depth2 * expansion
        self.conv_depth3 = conv1x1(depth2, depth3, 2)
        self.norm_depth3 = nn.BatchNorm2d(depth3)
        self.layer3 = nn.Sequential(
            *[ConvBlock(depth2, depth3, nn.BatchNorm2d)] * nr_layer_blocks
        )
        depth4 = depth3 * expansion
        self.conv_depth4 = conv1x1(depth3, depth4, 2)
        self.norm_depth4 = nn.BatchNorm2d(depth4)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.classification_layer = nn.Sequential(*[
            # nn.Dropout(0.3),
            nn.Linear(depth4, depth2),
            nn.Dropout(0.3),
            nn.Linear(depth2, num_classes)
        ])
    def forward(self, x):
        x = self.layer1(self.norm_depth1(self.conv_depth1(self.relu(self.conv1(x)))))
        x = self.layer2(self.norm_depth2(self.relu(self.conv_depth2(x))))
        x = self.layer3(self.norm_depth3(self.relu(self.conv_depth3(x))))
        x = self.avgpool(self.norm_depth4(self.relu(self.conv_depth4(x))))
        return self.classification_layer(torch.flatten(x, 1))
```

Pentru modelul cu augmentari aplicate putem observa ca learning rate-ul initial este mai mare decat ar trebui si duce la variatii in graficul pentru loss, ulterior, valoarea acestuia scade datorita learning scheduler-ului, iar curbele

de pierdere atat pentru training cat si pentru validare urmeaza aceeasi traiectorie, observam ca in ultimile epoci pierderea pentru validation incepe din nou sa scada, sugerand ca modelul ar fi putut fi antrenat pentru mai mult de 50 de epoci. In cazul acuratetii, indiferent de modificarile aplicate arhitecturii ca de exemplu crescand numarul de straturi, sau modificand parametrul de expansion pentru a face arhitectura "mai lata", acuratetea nu depaseste 80%, cu toate acestea antrenareaa este stabila, graficele atat pentru antrenare cat si pentru validare urmand aceeasi traiectorie, la fel ca si in graficul de loss, validare pare sa aiba o tendinta crescatoare, sugerand ca modelul ar fi putut fi antrenat pentru mai mult de 50 de epoci. Pentru modelul fara augmentari pe datele de transformare



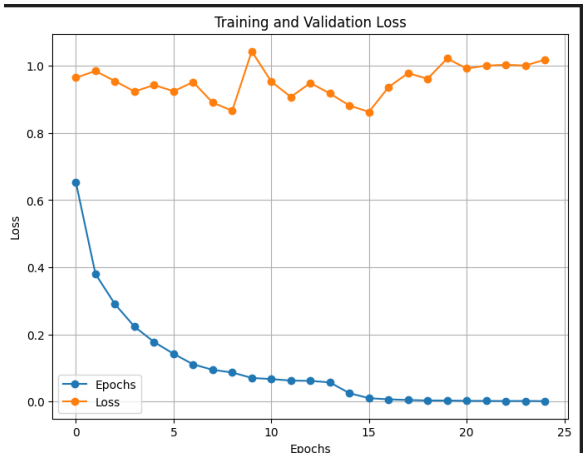
(a) Loss



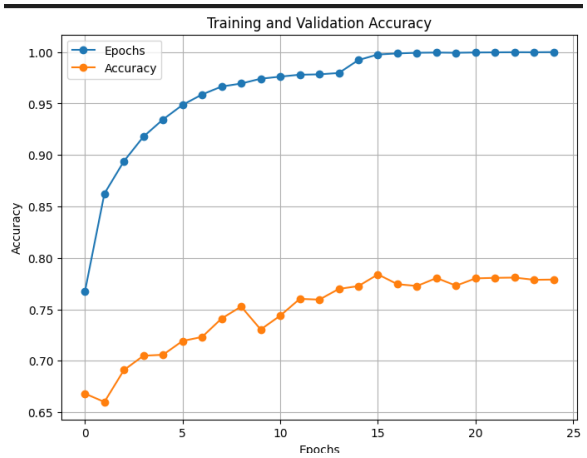
(b) Acuratete

Figure 5: Pierderea si acuratetea pe FashionMNIST aplicand arhitectura convolutionala cu augmentari

putem observa ca s-a obtinut un overfiting, pierderea pe datele de antrenare este 0 si acuratetea 1, sugerand ca modelul a invatat pe de rost training setul, iar eroarea pe validare ramane stabila pe toata durata antrenarii. In cazul acuratetii, modelul atinge o acuratete de pana in 80%, dar in ultimile 10 epoci valoarea nu se modifica.



(a) Loss



(b) Acuratete

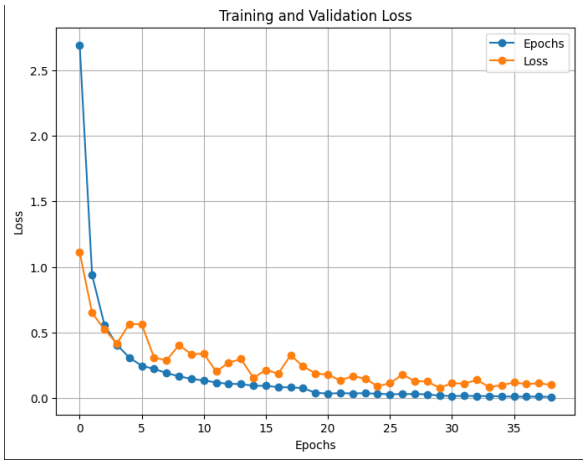
Figure 6: Pierderea si acuratetea pe FashionMNIST aplicand arhitectura convolutionala fara augmentari pe datele de antrenament

3.3 Rezultate Fruits

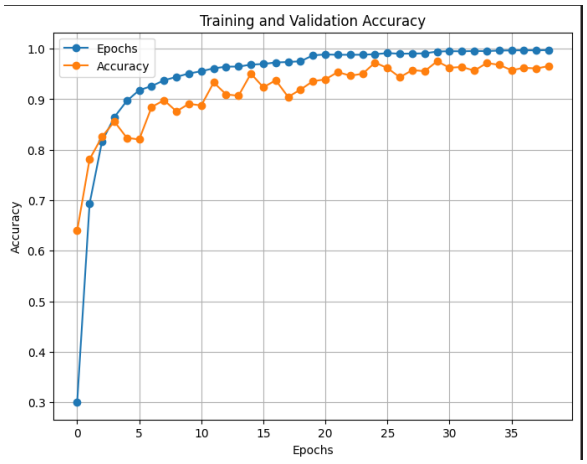
- batch size este de 100 si nummarul de epoci este 50, dintre care s-au executat doar 38
- functia de pierdere este CrossEntropyLoss si optimizatorul este Adamm cu learning rate-ul initial 1e-4 si 'weight_decay_factor': 1e-4
- modific learning rate-ul modelului folosind ReduceLROnPlateau avand un patience de 3, si modificarea valorii este de 0.5, lr = lr * 0.5

- pentru a preveni overfitting folosesc o strategie de early stopping cu patience de 9
- parametrii aditionali pe care ii folosesc pentru a modifica arhitectura initial a modelului convolutional sunt: nr_layers_block = 2, base_depth = 16, expansion = 4, si numarul initial de canale este 3, fiindca imaginile sunt in format color.
- ca metode de regularizare s-au folosit BatchNormalization2d, Dropout cu probabilitatea de p = 0.3 in stratul de clasificare, si strategia de early stopping
- modelul este alcatuit din straturi convolutionale, iar ultimul layer este cel de clasificare si este alcatuit din straturi liniare, modelul are 10173181 parametrii
- rezultate: acuratetea: 91%, f1 score: 85%, recall: 88%, precizie: 87%, antrenarea a durat 32 de minute si 39 de secunde

In situatia in care s-au aplicat augmentari pe datele de intrare am obtinut. Observam ca loss scade uniform pentru datele de antrenare ajungand destul dupa primele 20 de epoci la 0, fiindca learning rate-ul scade cu cateva ordine de marime, variatiile parametrilor devin mult prea mici pentru a influenta rezultatul general al modelului, observam ca atat graficul pentru loss pentru antrenare si validare, cat si graficele pentru acuratete sunt apropiate de valori una de cealalta, sugerand o antrenare stabila, iar spike-urile in setul de validare sunt reduse atat in graficul pentru loss cat si pentru acuratete, sugerand ca learning rate ul ales este unul potrivit.



(a) Loss



(b) Acuratete

Figure 7: Graficele de loss si de acuratete folosind modelul convolutional cu augmentari pe imagini pe dataset-ul Fruits

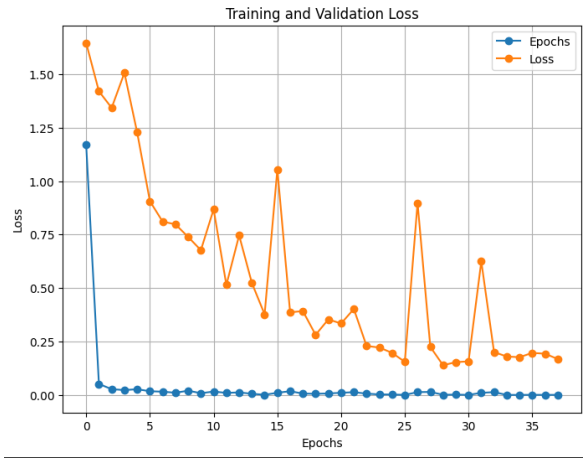
In situatia in care imaginile din setul de antrenare nu trec prin nicio forma de augmentare, aplicandu-se doar ToTensor() si Resize((28,28)). Modelul ajunge sa faca overfitting, eroarea la antrenare scade la 0 dupa primele 2 sau 3 epoci, iar eroarea la validare fluctueaza masiv, sugerand un posibil overfitting, acelasi lucru este aratat si de graficul pentru acuratete, cu toate acestea modelul ajunge la acuratete foarte mare pe setul de validare, iar aceasta ramane multumitoare pe setul de test, lipsa augmentarilor duce la overfitting.

- rezultatele obtinute sunt: acuratete: 86%, f1 score: 83%, recall: 85%, precizie: 83%, antrenarea a durat 17 min si 58 de secunde
- singurele transformari aplicate au fost ToTensor(), si Resize((28, 28))

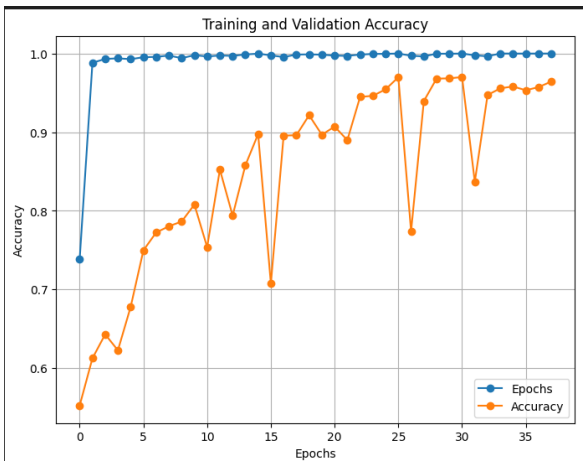
4 Utilizarea unei proceduri de finetuning peste arhitectura ResNet-18

4.1 Rezultate FashionMNIST

- batch size de 100, am facut resize al imaginilor din FashionMNIST la (32, 32) pentru a se putea protrivi cu arhitectura modelului de resnet, am obtinut modelul preantrenat pe datele din Cifar-10 preluand de aceasta adresa PyTorch CIFAR-10 GitHub.



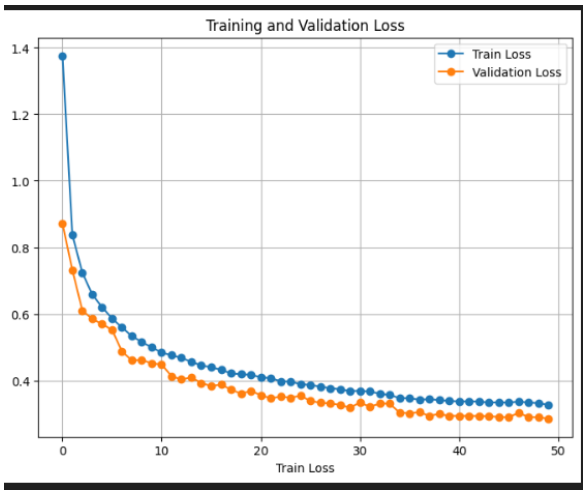
(a) Loss



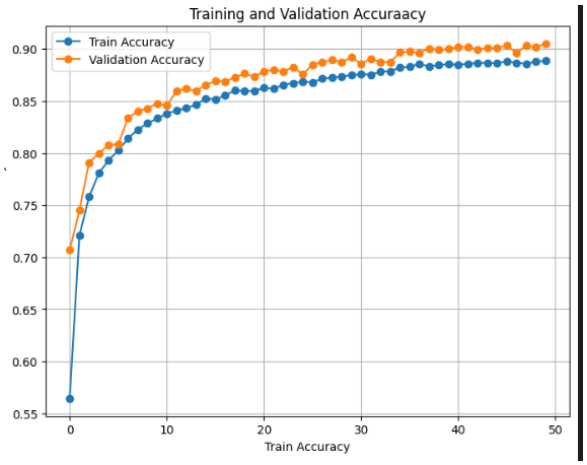
(b) Acuratete

Figure 8: Graficele de loss si de acuratete folosind modelul convolutional fara augmentari pe imagini, pe dataset-ul Fruits

- ca functii de augmentare pentru datele de antrenare am folosit: RandomHorizontalFlip, RandomRotation(degrees=90), RandomCrop(size=((32, 32), padding=(4, 4), padding_mode="reflect"), Normalize(mean = [0.4914, 0.4822, 0.4465], std = [0.2471, 0.2435, 0.2616]), iar atat pe datele de antrenare, validare si testare am aplicat, ToTensor(), Resize((32,32) si Normalize(mean = [0.4914, 0.4822, 0.4465], std = [0.2471, 0.2435, 0.2616])
- am antrenat pentru 50 de epoci, ca functie de pierdere am folosit CrossEntropyLoss(), si ca optimizator am folosit torch.optim.SGD cu learning rate initial de 1e-4, si weight_decay_factor de 1e-4 si momentul de 0.5
- folosesc un learning rate scheduler ReduceLROnPlateau, cu factor de reduce de 0.4 si patience de 3
- folosesc un early stopper cu patience de 5, dar in cele 50 de epoci modelul a avansat constant si nu a fost nevoie de acesta
- antrenarea a durat 28 de minute, si am obtinut acuratetea 89%, f1: 89% 'recall': 89%, precision: 89%



(a) Loss



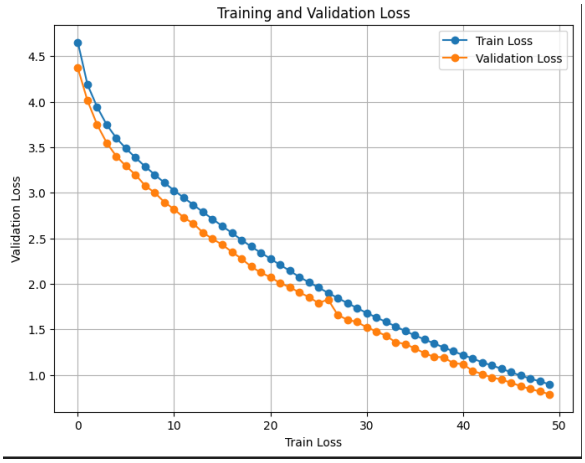
(b) Acuratete

Figure 9: Pierderea si acuratetea pe FashionMNIST aplicand o arhitectura preantrenata de tip Resnet18

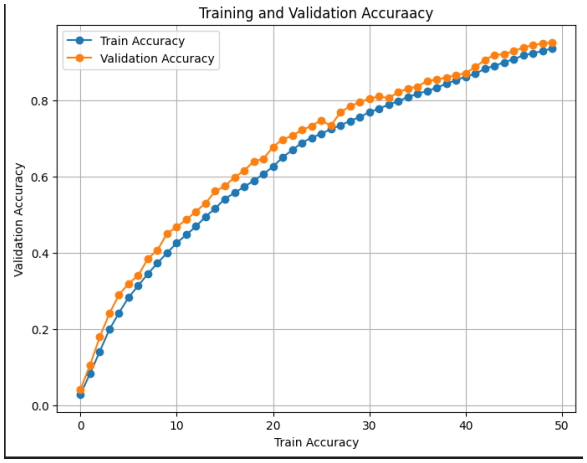
Pierderea atat pentru train cat si pentru validation scade uniform, fara a avea spike-uri, sugerand un model stabil, avand un learning rate potrivit, de asemenea, modelul pare sa specializat mai mult pe a generaliza, desurcandu-se mai bine pe date necunoscute decat pe setul de antrenare.

4.2 Rezultate Fruits

- batch size de 100, am facut resize al imaginilor din Fruits la (32, 32) pentru a se putea protrivi cu arhitectura modelului de resnet, am obtinut modelul preantrenat pe datele din Cifar-10 preluand de aceasta adresa PyTorch CIFAR-10 GitHub.
- ca functii de augmentare pentru datele de antrenare am folosit: RandomHorizontalFlip, RandomRotation(degrees=90), RandomCrop(size=((32, 32), padding=(4, 4), padding_mode="reflect"), Normalize(mean = [0.4914, 0.4822, 0.4465], std = [0.2471, 0.2435, 0.2616]), iar atat pe datele de antrenare, validare si testare am aplicat, ToTensor(), Resize((32,32) si Normalize(mean = [0.4914, 0.4822, 0.4465], std = [0.2471, 0.2435, 0.2616])
- am antrenat pentru 50 de epoci, ca functie de pierdere am folosit CrossEntropyLoss(), si ca optimizator am folosit torch.optim.SGD cu learning rate initial de 1e-4, si weight_decay_factor de 1e-4 si momentul de 0.5
- folosesc un learning rate scheduler ReduceLROnPlateau, cu factor de reduce de 0.4 si patience de 3
- folosesc un early stopper cu patience de 5, dar in cele 50 de epoci modelul a avansat constant si nu a fost nevoie de acesta
- antrenarea a durat 41 de minute, si am obtinut acuratetea 92%, f1: 89% 'recall': 89%, precision: 95%



(a) Loss



(b) Acuratete

Figure 10: Pierderea si acuratetea pe Fruits aplicand o arhitectura preantrenata de tip Resnet18

Chiar daca am antrenat pentru 50 de epoci, evolutia graficului atat pentru eroare cat si pentru validare au ramas stabile, sugerand ca modelul ar mai fi putut fi antrenat macar 10 epoci pentru a maxima acuratatea acestuia.

5 Raport final

5.1 Concluzii

- cel mai probabil ceva este gresit in modul in care fac extragerea de feature-uri pentru task-ul 1, acuratatea obtina de modele MLP aplicate pe feature-uri fiind neasteptat de scazuta
- Aplicarea augmentării imaginilor a îmbunătățit semnificativ performanța modelului în toate abordările si reduce overfiting-ul
- Modelele convoluționale si cele fine-tuned precum Resnet18, au avut rezultate mai bune în comparație cu modelele simple (MLP), datorită capacității lor de a extrage informații complexe din date.
- modelele fine-tuned sunt mai stabile in procesul antrenare, nu exista spike-uri in graficul pentru acuratete sau pierdere pentru validare, aratand ca modelul generalizeaza bine pe tot parcursul antrenarii
- modelele de tip MLP se antreneaza cel mai rapid, dar tind sa faca overfitting, modele convolutionale dureaza mai mult ca timp pentru antrenare, dar tind sa generalizeze mai bine iar antrenarea este mai stabila, aceasta converge mai repede catre o solutie optima.

Model	Acuratețe (%)	F1 Score (%)	Recall (%)	Precizie (%)	Timp (min)
FashionMNIST (features)	64.00	64	64	64	3
FashionMNIST (mlp)	88	88	88	88	4
FashionMNIST (conv cu augmentări)	81.94	81	81	82	24
FashionMNIST (conv fără augmentări)	77.00	78	77	80	17
ResNet-18 Fine-tuning (FashionMNIST)	89.00	89	89	89	28
Fruits (features)	29	16	17	18	7
Fruits (mlp)	95	92	92	94	7
Fruits (cu augmentări)	91.00	85	88	87	32
Fruits (fără augmentări)	86.00	83	85	83	18
ResNet-18 Fine-tuning (Fruits)	92.00	89	89	95	41

Table 1: Rezultatele obținute pentru diferite modele, incluzând acuratețea, F1 score, recall, precizia și timpul de execuție.

- pentru a pastra un model de tip MLP stabil este necesar ca rata acestuia de invatare sa fie foarte mica, undeva la 1e-5.