



UNIVERSITATEA DIN BUCUREŞTI  
FACULTATEA DE MATEMATICĂ ŞI  
INFORMATICĂ



SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENȚĂ

---

Sistem pentru detecția și clasificarea  
mașinilor din imagini în funcție de model

COORDONATOR ȘTIINȚIFIC  
PROF. DR. RADU IONESCU

STUDENT  
ANDREI-CRISTIAN GÎDEA

BUCUREŞTI, ROMÂNIA  
IULIE 2020

## Rezumat

Dezvoltarea accelerată a domeniului ce se ocupă de *deep learning* nu poate fi neglijată. Vederea artificială sau *computer vision*-ul este un subdomeniu al *deep learning*-ului ce se ocupă cu dezvoltarea aplicațiilor ce oferă dispozitivelor capabilitățile necesare pentru înțelegerea mediului înconjurător prin prelucrarea imaginilor. Printre aplicațiile practice ale vederii artificiale se numără atât detecția obiectelor în imagini, cât și clasificarea imaginilor în funcție de anumite clase.

Prin prezenta lucrare de licență, cu numele "Sistem pentru detectia și clasificarea mașinilor din imagini în funcție de model" se propune o soluție pentru detectarea și recunoașterea modelelor de mașini din imagini folosind elemente specifice *deep learning*-ului. Astfel, industriile din domeniul auto pot beneficia de pe urma acestei implementări, putând fi micșorat timpul alocat anumitor sarcini. În continuare sunt prezentate abordările și procesele necesare în implementarea acestei soluții care primește o imagine și pe baza ei detectează mașinile existente, ca mai apoi să le clasifice în funcție de model. Pentru partea de detectie de mașini am folosit un algoritm clasic, Faster R-CNN. Rețeaua convinguțională de bază care se ocupă cu extragerea caracteristicilor pentru acesta am ales să fie ResNet18 datorită numărului relativ mic de parametri și al performanțelor bune obținute. Pentru sarcina de clasificare a acestor detectii am ales să folosesc o rețea mai puternică, ResNet34, fiind alegerea ideală din punctul de vedere al performanței/timpului alocat pentru o clasificare. Interfața web, ce are scopuri demonstrative, am ales să o dezvolt folosind Flask, acesta facilitând procesul de "servire" al aplicației de bază(sistemul detector + clasificator).

Sistemul obținut detectează mașinile prezente în imagini, însă nu tot timpul, având probleme cu mașinile de dimensiuni mici, deoarece setul de date pe care detectorul a fost antrenat nu conține asemenea exemple de antrenare. În plus, clasificatorul a obținut niște rezultate foarte bune, cu o eroare de doar 6.63% dacă ne luăm după prima predicție, sau 0.83% dacă ne luăm după primele cinci predicții făcute, ceea ce, în opinia mea, depășește capacitatea unei persoane de recunoaștere a unei mașini după model.

Astfel, conform rezultatelor, aplicația poate fi folosită cu succes în diverse scenarii ce necesită lucrul cu mașini și recunoașterea acestora.

## Abstract

The accelerated development of the field dealing with deep learning cannot be neglected. Artificial vision or computer vision is a subdomain of deep learning that deals with the development of applications that provide devices with the necessary capabilities to understand the environment through image processing. Practical applications of artificial vision include both object detection in images and image classification according to certain classes. This license paper, entitled "Car Models Detection and Classification System", proposes a solution for detecting and recognizing cars models in images using specific elements of deep learning. Thus, the automotive industries can benefit from this implementation, and the time allocated to certain tasks can be reduced. Below are the approaches and processes required in implementing this solution that receives an image and based on it detects existing cars, and then classifies them according to model. For the car detection part I used a classic algorithm, Faster R-CNN. The basic convolutional network that deals with the extraction of the characteristics for it I chose to be ResNet18 due to the relatively small number of parameters and the good performances obtained. For the classification task of these detections I chose to use a stronger network, ResNet34, being the ideal choice in terms of performance / time allocated for a classification. The web interface, which has demonstration purposes, I chose to develop it using Flask, which facilitates the process of "serving" the basic application (detector + classifier system).

The obtained system detects the cars present in the images, but not all the time, having problems with the small cars, because the data set on which the detector was trained does not contain such training examples. In addition, the classifier gave very good results, with an error of only 6.63% if we take it after the first prediction, or 0.83% if we take it after the first five predictions obtained, which, in my opinion, exceeds a person's ability. of recognizing a car by model.

Thus, according to the results, the application can be used successfully in various scenarios that require working with cars and their recognition.

# Cuprins

## Listă de figuri

## Listă de tabele

<b>1</b>	<b>Introducere</b>	<b>1</b>
1.1	Scopul și motivația alegerii temei . . . . .	1
1.2	Structura lucrării . . . . .	2
<b>2</b>	<b>Concepțe teoretice</b>	<b>3</b>
2.1	Rețele convoluționale . . . . .	3
2.1.1	Tipuri de straturi . . . . .	4
2.2	Antrenarea rețelei . . . . .	7
2.2.1	Algoritmul de <i>backpropagare</i> . . . . .	7
2.2.2	Optimizare . . . . .	8
2.3	<i>Transfer learning</i> . . . . .	9
2.4	<i>Hook</i> . . . . .	11
2.5	Concepțe necesare în evaluarea unei rețele . . . . .	11
<b>3</b>	<b>Abordări recente</b>	<b>13</b>
<b>4</b>	<b>Tehnologii folosite</b>	<b>15</b>
4.1	Python . . . . .	15
4.1.1	Aplicația de bază . . . . .	15
4.1.2	NumPy . . . . .	15
4.1.3	Pytorch . . . . .	16
4.1.4	OpenCV . . . . .	16
4.1.5	PIL . . . . .	16

4.1.6	scikit-learn	16
4.1.7	pandas	17
4.2	Interfață web	17
4.2.1	Flask	17
4.2.2	Bootstrap	17
<b>5</b>	<b>Abordare propusă</b>	<b>18</b>
5.1	Faster R-CNN(RPN + Fast R-CNN)	19
5.1.1	Cum funcționează Faster R-CNN	20
5.1.2	RPN	21
5.1.3	Fast R-CNN	23
5.1.4	Antrenare	27
5.2	ResNet	27
5.2.1	Prima conoluție	31
5.2.2	<i>Layerele</i> din ResNet	32
5.2.3	Tipare în construcția <i>ResNet</i> -ului	33
5.2.4	Mecanism de atenție	36
5.2.5	Antrenare	38
5.3	Evaluarea metodei	38
5.3.1	<i>Grad-CAM</i>	39
<b>6</b>	<b>Experimente și rezultate</b>	<b>41</b>
6.1	Pregătirea datelor	41
6.2	Detectia de mașini	44
6.2.1	Grafice antrenament	45
6.3	Clasificarea modelelor de mașini	46
6.3.1	Grafice antrenament/validare	49
<b>7</b>	<b>Descrierea aplicației</b>	<b>50</b>
7.1	Interfață web	50
7.2	Aplicația de bază	52
<b>8</b>	<b>Concluzii și dezvoltări ulterioare</b>	<b>53</b>

# Listă de figuri

2.1	Modul de funcționare al unui perceptron. Figură preluată din [6] . . . . .	4
2.2	Un exemplu de volum de intrare(o imagine CIFAR-10 32x32x3) și un volum de neuroni în primul strat conoluțional. Fiecare neuron din stratul conoluțional este conectat doar la o regiune în volumul de intrare. Figură preluată din [3] . . . . .	5
2.3	Activări ale straturilor conoluționale din ResNet34. . . . .	5
2.4	Batch normalization aplicat pe o activare $x$ peste un mini-batch. Figură preluată din [16]. . . . .	6
2.5	Punct de minim local și punct "șa". Figură preluată din [3] . . . . .	8
2.6	Modificarea ponderilor prin <i>momentum</i> . Figură preluată din [3] . . . . .	9
2.7	Precizia și recall-ul . . . . .	12
3.1	Evoluția detectoarelor de obiecte. Printre cele mai importante se numără R-CNN[12], Fast R-CNN[11], Faster R-CNN[30], YOLO[29], SSD[20]. Figură preluată din [44] . . . . .	13
5.1	Arhitectura pentru Faster R-CNN. Figură preluată din [30] . . . . .	19
5.2	Arhitectura pentru RPN. . . . .	20
5.3	Ancore la pozitia (600, 600). Fiecare dimensiune este reprezentată în altă culoare. . . . .	22
5.4	Arhitectura Faster R-CNN detaliată. Figură inspirată din [1] și [28]. . . . .	23
5.5	Maparea RoI-ului. . . . .	25
5.6	<b>RoIAlign:</b> liniile punctate reprezintă un <i>feature map</i> , liniile solide un RoI(cu celule de dimensiune 2x2) și punctele marcate reprezintă cele 4 puncte de eșantionare din fiecare celulă. RoIAlign calculează valoarea fiecărui punct de eșantionare prin interpolare biliniară folosind punctele din apropiere de pe <i>feature map</i> . Nu se efectuează cuantificări deloc. Figură preluată din [14]	25

5.7	Ecuația interpolării biliniare. Figură preluată din [40] . . . . .	26
5.8	Distribuția celor 4 puncte dintr-o celulă. . . . .	26
5.9	Interpolarea biliniară pentru primul punct . . . . .	26
5.10	Arhitectura cu 56 de straturi are performanțe mai proaste atât pe antrenare cât și pe testare față de arhitectura cu 20 de straturi. Figură preluată din [3] .	28
5.11	Bloc rezidual. Figură preluată din [13] . . . . .	28
5.12	O funcție reziduală mai adâncă $\mathcal{F}$ pentru ImageNet. Stânga: un bloc de bază sau <i>BasicBlock</i> (pe feature map-uri de 56x56) ca în Figura 5.13 pentru ResNet34. Dreapta: un <i>Bottleneck</i> pentru ResNet-50/101/152. Figură preluată din [13] . . . . .	29
5.13	Exemple de arhitecturi pentru ImageNet. Stânga: Modelul VGG-19 [34]. Mijloc: o rețea simplă cu 34 de straturi. Dreapta: o rețea reziduală cu 34 de straturi. Tabelul 5.1 prezintă mai multe detalii și alte variante. Figură preluată din [13] . . . . .	30
5.14	conv1 - Stratul de conoluție. Figură preluată din [1] . . . . .	31
5.15	conv1 - <i>max pooling</i> . Figură preluată din [1] . . . . .	31
5.16	Primul <i>layer</i> , primul bloc, prima operație. Figură preluată din [1] . . . . .	32
5.17	Primul <i>layer</i> , primul bloc. Figură preluată din [1] . . . . .	33
5.18	Primul <i>layer</i> . Figură preluată din [1] . . . . .	33
5.19	Al doilea <i>layer</i> , primul bloc, prima operație. Figură preluată din [1] . . . . .	34
5.20	<i>Projection Shortcut</i> . Figură preluată din [1] . . . . .	34
5.21	Al doilea <i>layer</i> , primul bloc. Figură preluată din [1] . . . . .	34
5.22	Al doilea <i>layer</i> . Figură preluată din [1] . . . . .	35
5.23	Un bloc <i>Squeeze and excitation</i> . Figură preluată din [15] . . . . .	36
5.24	Stânga: un bloc rezidual. Dreapta: un modul SEResNet folosit și în prezența lucrare de licență. Figură preluată din [15] . . . . .	37
5.25	Prezentare generală a CBAM-ului. Modulul are două sub-module secvențiale: pe adâncimea canalelor și pe dimensiunea spațială. Figură preluată din [42]	37

5.26 Diagrama fiecărui submodul de atenție. Submodulul pentru canale folosește atât ieșiri de <i>max-pooling</i> , cât și ieșiri de <i>avg-pooling</i> cu o rețea partajată; submodulul spațial utilizează două ieșiri similare, care sunt concatenate de-a lungul axei canalelor și introduse într-un strat convecțional. Figură preluată din [42] . . . . .	38
5.27 CBAM integrat cu un bloc rezidual în ResNet[13]. Figură preluată din [42]	38
5.28 Modul de funcționare al <i>Grad-CAM</i> -ului. Figură inspirată din [33] . . . . .	39
5.29 <i>Heatmap</i> generat de Grad-CAM . . . . .	40
6.1 Reprezentarea bounding-box-urilor . . . . .	42
6.2 Augmentare pentru detector. . . . .	43
6.3 Diverse transformări . . . . .	43
6.4 Problemele detectorului cu mașinile în dimensiuni mici. . . . .	45
6.5 Loss-ul regresiei din RPN pe datele de antrenament . . . . .	45
6.6 Loss-ul clasificatorului din RPN pe datele de antrenament . . . . .	45
6.7 Loss-ul regresiei pe bounding box-uri pe datele de antrenament . . . . .	45
6.8 Loss-ul clasificatorului pe datele de antrenament . . . . .	45
6.9 Loss-ul total pe datele de antrenament . . . . .	46
6.10 Exemplu din clasa Aston Martin Virage Convertible 2012 clasificat greșit ca fiind Aston Martin V8 Vantage Coupe 2012 de către ResNet34. . . . .	47
6.11 Exemplu din clasa Aston Martin Virage Convertible 2012 clasificat greșit ca fiind Aston Martin V8 Vantage Coupe 2012 de către SEResNet34 . . . . .	47
6.12 Exemplu din clasa Aston Martin Virage Convertible 2012 clasificat greșit ca fiind Aston Martin V8 Vantage Coupe 2012 de către CBAMResNet34 . . . . .	48
6.13 Modele cu care modelul din Figurile 6.10, 6.11 și 6.12 poate fi confundat ușor, chiar și de o persoană fără prea multă experiență. . . . .	48
6.14 Acuratețea pe datele de antrenament ResNet34 . . . . .	49
6.15 Loss-ul pe datele de antrenament ResNet34 . . . . .	49
6.16 Acuratețea pe datele de validare ResNet34 . . . . .	49
6.17 Loss-ul pe datele de validare ResNet34 . . . . .	49
7.1 Interfața web în momentul primei intrări. . . . .	51
7.2 Posibilitatea de a schimba modelul folosit în clasificare. . . . .	51

7.3	Modelele disponibile sunt: ResNet34, SEResNet34 si CBAMResNet34. . .	51
7.4	Schimbarea modelului folosit pentru clasificare. . . . .	51
7.5	Detectiile de mașini. . . . .	51
7.6	Clasificarea unei mașini în funcție de model. . . . .	52
7.7	<i>Heatmap</i> -ul unei clasificări. . . . .	52
7.8	În partea stangă a paginii: rezultatele clasificării. În partea dreaptă: top-ul predicțiilor ordonate descrescător. . . . .	52

# Listă de tabele

5.1 Arhitecturi ResNet pentru ImageNet. Blocurile de construcție sunt prezentate între paranteze (vezi și Figura 5.12), cu numărul de blocuri stivuite. Downsampling-ul este realizat de conv3_1, conv4_1 și conv5_1 cu un pas(stride) de 2. Tabel inspirat din [13] . . . . .	35
6.1 <i>mAP</i> -ul pentru detectorul folosind ResNet18 cu straturile înghețate vs folosind ResNet18 care își schimbă ponderile odată cu antrenarea acestuia. . . . .	44
6.2 Variația <i>mAP</i> -ului în funcție de dimensiunea imaginii de intrare și aspectul ancorelor. . . . .	44
6.3 <i>mAP</i> pentru Metoda 1 vs Metoda 2, descrise în Capitolul 5 . . . . .	44
6.4 Rata de eroare(%) pentru subsetul de testare al setului de date. "FE" în seamnă <i>feature-extractor</i> (detalii în Secțiunea ??). . . . .	48
6.5 Rata de eroare(%) pentru subsetul de testare al setului de date. . . . .	48
6.6 Acuratețea(%)medie pentru subsetul de testare al setului de date și acuratețea pe unele clase. . . . .	49

# Capitolul 1

## Introducere

Deși inteligența artificială există de foarte mult timp, aceasta, în special domeniul *deep learning*-ului, a intrat în atenția publicului începând cu anul 2012, când competiția ImageNet[5] a fost câștigată de o rețea conoluțională adâncă, intitulată AlexNet[19]. Acest lucru a fost posibil abia în momentul acela deoarece până atunci puterea computațională era destul de redusă, modelele adânci având nevoie de foarte multe calcule pentru a învăța. Astfel, odată cu dezvoltarea hardware-ului, plăcile video, unde se întâmplă computațiile, au devenit din ce în ce mai puternice, favorizând crearea de framework-uri destinate *deep learning*-ului care facilitează scrierea și crearea de noi modele. Printre cele mai cunoscute astfel de framework-uri se numără Pytorch, Caffe, Tensorflow și multe altele.

### 1.1 Scopul și motivația alegerii temei

Odată cu dezvoltarea tehnologiilor ce permit ca vederea artificială să fie folosită pe o varietate mare de dispozitive, aceasta nu mai poate fi privită ca un lucru neobișnuit, ci ca pe ceva normal. Am ales să dezvolt această temă cu scopul de a accelera unele procese din domenii care se ocupă cu mașini. De exemplu, aplicația de bază poate fi foarte ușor integrată într-un site care se ocupă cu vânzări auto, făcând automat procesul de introducere a datelor despre mașină(marca, modelul, anul fabricației, tipul caroseriei) doar introducând o poză cu mașina respectivă. De asemenea, aplicația de bază poate fi rulată pe camerele video de supraveghere a traficului, astfel că în momentul când o mașină este furată și căutată de către poliție, programul să micșoreze semnificativ timpul de căutare printre imagini, filtrând și căutând doar în acele imagini care conțin mașini aparținând aceluiași model(apoi putând fi

aplicat un algoritm pentru recunoașterea numerelor de înmatriculare, de exemplu).

## 1.2 Structura lucrării

**Aprofundarea conceptelor teoretice și tehnologice.** În Capitolul 2 sunt prezentate toate concepțele teoretice cunoscute din domeniul vederii artificiale, ce sunt necesare pentru dezvoltarea soluției prezentate. Aceste concepte sunt aplicate cu ajutorul mai multor tehnologii(Python, PyTorch, Flask), tehnologii ce sunt detaliate în Capitolul 4.

**Dezvoltarea aplicativă și îmbunătățirea soluției.** În Capitolul 5 sunt expuse toate abordările testate precum și alegera celei care a dat cele mai bune rezultate. În plus, sunt expuși algoritmii adaptați și folosiți pentru rezolvarea problemei. În Capitolul 6 rezultatele însotite de tabele și figuri ilustrative sunt expuse pentru o mai bună înțelegere a dificultăților întâlnite în parcurgerea rezolvării acestei probleme. De asemenea, Secțiunea 6.1 a acestui capitol prezintă în detaliu procesul ce privește colectarea datelor, curățarea acestora și pregătirea pentru a putea fi folosite în contextul actual. Capitolul 7 descrie atât aplicația de bază, care se ocupă cu detecția și clasificarea mașinilor din imagini, cât și aplicația ce are rol de interfață web, prin care utilizatorul are acces la aplicația principală. În Capitolul 8 îmi expun atât concluziile la care am ajuns lucrând la această temă de licență, cât și dezvoltările sau îmbunătățirile ulterioare ce pot fi aduse.

# **Capitolul 2**

## **Concepțe teoretice**

### **2.1 Rețele convoluționale**

O rețea neuronală convoluțională(ConvNet sau CNN) reprezintă un algoritm folosit în *deep learning* ce acceptă un volum de intrare(de obicei sunt folosite pentru imagini, ca și în cazul prezentei lucrări) căruia îi atribuie niște ponderi ce au ca scop diferențierea diverselor caracteristici ale acesteia. Față de metodele precedente, unde caracteristicile trebuiau concepute manual de cineva cu experiență, CNN-urile au capacitatea de a le învăța. Arhitectura unei rețele convoluționale seamănă foarte mult cu conectivitatea neuronilor din creierul uman, fiind inspirată de cortexul vizual. Aici, neuronii răspund doar la stimuli veniți dintr-o zonă restrânsă a câmpului vizual denumit câmp receptiv(echivalent, acesta reprezintă dimensiunea unui filtru folosit în convoluții). Prin alăturarea acestor neuroni, întreaga zonă vizuală este acoperită și astfel imaginea este procesată. Perceptronul utilizat în rețelele neuronale a rămas la fel și în cazul celor convoluționale. Acesta reprezintă unitatea de bază a unei rețele neuronale, fiind inspirat de neuronii și legăturile ce se creează între aceștia în creierul uman. El poate primi la intrare fie datele initiale, fie rezultatele produse de perceptronii precedenți. În urma unor calcule, acesta produce o valoare ce este dată mai departe straturilor superioare. Dacă intrarea este notată cu litera  $X$ , sub forma unei matrici și ponderile ce trebuie învățate, specifice fiecărui perceptron, cu  $W$ , tot sub forma unei matrici, atunci perceptronul va face produsul scalar dintre acestea. Rezultatului astfel obținut îi este adăugat un *bias* și apoi îi este aplicată o funcție de activare. Funcțiile de activare sunt importante în rețelele neuronale deoarece acestea adaugă non-liniaritate. Astfel, rezultatul

final poate fi scris și sub forma: activare( $X * W + bias$ ) (vezi Figura 2.1). În ResNet, singura funcție de activare folosită este ReLU, prescurtare ce vine de la *rectified linear unit*, descrisă prin Ecuația 2.1.

$$f(x) = \max(0, x) \quad (2.1)$$

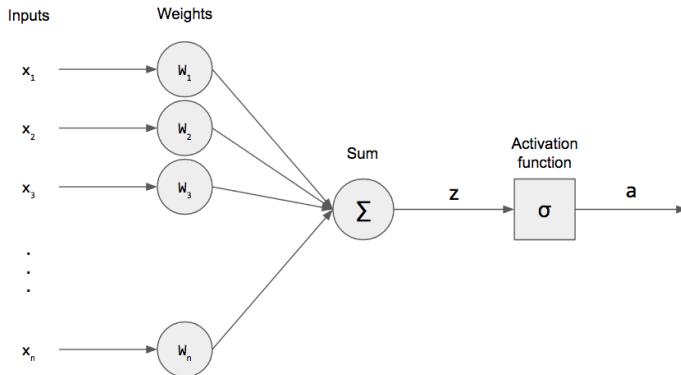


Figura 2.1: Modul de funcționare al unui perceptron. Figură preluată din [6]

O rețea conoluțională este formată din mai mulți neuroni organizati în straturi conectate între ele. Pentru a antrena acești neuroni, se folosește algoritmul de *backpropagare*, detaliat în Secțiunea 2.2.

### 2.1.1 Tipuri de straturi

În continuare voi prezenta toate straturile ce au fost necesare în implementarea aplicației, luând în considerare că arhitectura ce stă la baza detectorului de mașini 5.1 cât și la cea a clasificatorului în funcție de model este rețeaua ResNet, descrisă mai în detaliu în 5.2.

#### Stratul conoluțional

Parametrii straturilor conoluționale sunt reprezentați de filtre ce pot fi învățate. De obicei filtrele folosite sunt de dimensiuni spațiale foarte mici, de exemplu 1x1 sau 3x3. Un astfel de filtru este glisat de-a lungul lățimii și înălțimii imaginii de intrare, făcându-se produsul scalar dintre cele două volume. Pe măsură ce filtrul este glisat, acesta va produce o hartă de activare (hartă de caracteristici sau *feature map*) care expune răspunsul aceluiași filtru la fiecare poziție. Se folosesc mai multe astfel de filtre, a căror ieșiri sunt concatenate, producând volumul de ieșire. Atunci când avem de-a face cu volume de intrare de dimensiuni

mari, cum ar fi imaginile, este foarte costisitor să folosim straturi *fully-connected*. Astfel, fiecare perceptron va fi legat la o mică regiune din intrare, după cum este expus în Figura 2.2 .

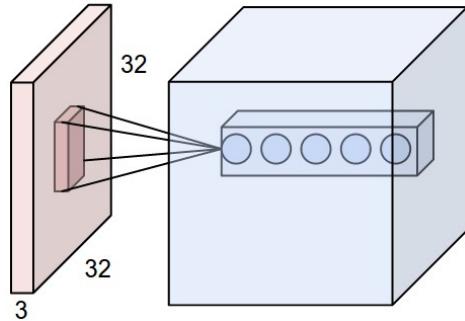


Figura 2.2: Un exemplu de volum de intrare(o imagine CIFAR-10 32x32x3) și un volum de neuroni în primul strat convolutional. Fiecare neuron din stratul convolutional este conectat doar la o regiune în volumul de intrare. Figură preluată din [3]

Dimensiunea spațială a ieșirii este calculată în funcție de dimensiunea volumului de intrare( $I$ ), dimensiunea filtrului( $K$ ), pasul sau *stride*-ul cu care înaintează filtrul( $S$ ) și cantitatea de pixeli bordați(*padded*) intrării( $P$ ). Formula este dată de ecuația:

$$\frac{I + 2 * P - K}{S} + 1$$

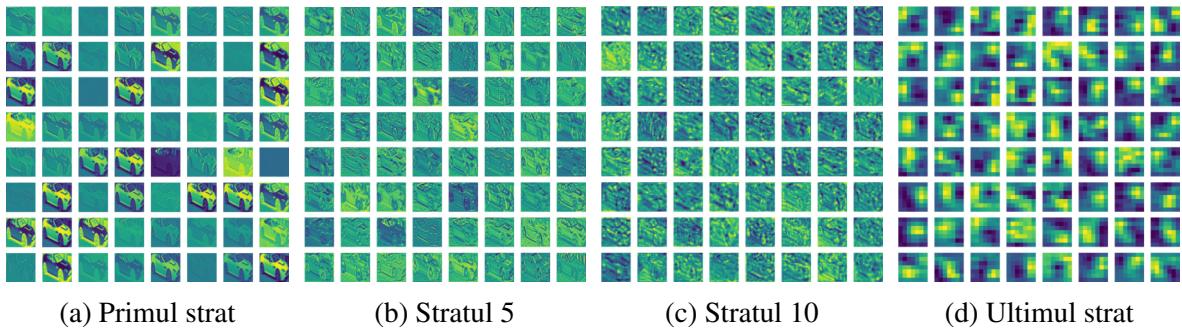


Figura 2.3: Activări ale straturilor convoluționale din ResNet34.

### Stratul de pooling

Funcția sa este de a reduce progresiv dimensiunea spațială a intrării. Acesta folosește doi parametrii, dimensiunea volumului de intrare( $I$ ) și pasul sau *stride*-ul cu care înaintează filtrul( $S$ ). Formula calculării dimensiunii ieșirii este asemănătoare cu cea din conoluție, cu diferența că nu mai există pixeli bordați.

## Stratul pentru normalizarea batch-ului

Normalizarea batch-ului("batch normalization" sau BatchNorm pe scurt) este un strat, ce prin doi parametrii antrenabili(asta însemnând că se antrenează împreună cu restul rețelei) încearcă ca fiecare mini-batch care trece prin rețea să aibă o medie centrală în jurul lui 0 și o varianță de aproximativ 1. În rețelele adânci, acesta rezolvă într-o oarecare măsură problema *vanishing gradient*-ului respectiv pe cea a *exploding gradient*-ului, apărute în urma înmulțirilor repetate, deoarece este o formă de regularizare. BatchNorm-ul poate fi interpretat și ca preprocesarea datelor la fiecare strat al rețelei.

Una dintre motivațiile principale pentru dezvoltarea BatchNorm[16] a fost reducerea aşa-numitei modificări a covariantei interne("internal covariate shift" sau ICS). În [16] autorii definesc ICS drept fenomenul în care distribuția intrărilor către un strat în rețea se schimbă datorită unei actualizări a parametrilor straturilor anterioare.

Optimizorul(SGD în acest caz) anulează această normalizare dacă este o modalitate de a minimiza funcția de pierdere. În consecință, normalizarea batch-ului adaugă doi parametrii antrenabili la fiecare strat, astfel încât ieșirea normalizată este înmulțită cu „deviația standard”(gamma) la care se adaugă „media”(beta)(daca  $\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$  și  $\beta^{(k)} = E[x^{(k)}]$ , atunci am putea recupera activările inițiale). Cu alte cuvinte, normalizarea batch-ului permite optimizorului să efectueze denormalizarea schimbând doar aceste două ponderi pentru fiecare activare, în loc să piardă stabilitatea rețelei modificând toate ponderile.

<b>Input:</b> Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$ ;
Parameters to be learned: $\gamma, \beta$
<b>Output:</b> $\{y_i = BN_{\gamma, \beta}(x_i)\}$
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$

Figura 2.4: Batch normalization aplicat pe o activare x peste un mini-batch. Figură preluată din [16].

Motivația pentru care acesta tehnica funcționează nu este încă 100% definitivă, drept

consecință, este un subiect pentru cercetare. În [32] autorii susțin că ICS-ul nu este direct responsabil cu performanța antrenării, cel puțin dacă ne referim la ICS prin stabilitatea mediei și varianței distribuțiilor intrărilor. Mai exact, aceștia au demonstrat că BatchNorm face spațiul corespunzător optimizării problemei mult mai "neted" (*smooth*), astfel permitând utilizarea unei rate de învățare mai mari și implicit accelerarea antrenării.

### Stratul *fully-connected*

Neuronii dintr-un strat *fully-connected* au conexiuni complete la toate activările din stratul anterior. Folosirea unui strat *fully-connected* este un mod bun de a învață combinațiile non-liniare ale caracteristicilor de nivel înalt. De aceea, de obicei în rețelele conveționale acestea se folosesc la sfârșitul ei pentru a scoate clasele dorite (în ResNet există un singur strat *fc* care mapează feature-urile scoase de straturile conveționale la numărul de clase dorite).

## 2.2 Antrenarea rețelei

### 2.2.1 Algoritmul de *backpropagare*

*Backpropagarea* sau *propagarea înapoi a erorilor* este un algoritm pentru antrenarea rețelelor neuronale folosind metoda coborârii pe gradient. Algoritm este format din doi pași și anume pasul de *forward* și cel de *backward*. După cum îi spune și numele, în *forward* rețeaua este aplicată unei intrări, iar cu ajutorul ponderilor actuale, aceasta face o predicție. În funcție de această predicție și de rezultatul ce trebuia obținut, este calculată valoarea ero- rii cu ajutorul funcției de *loss*. În pasul de *backward* se face calculul gradientului, doar că în sens invers. Mai exact, gradientul ultimului strat este calculat primul și propagat înapoi prin rețea până la primul strat. Astfel, parametrii rețelei sunt ajustați cu scopul de minimiza- rea eroare folosind algoritmi de optimizare. Cea mai cunoscută metodă pentru optimiza- rea rețelelor este coborârea pe gradient sau *gradient descent*, iar în această lucrare a fost utilizată una dintre variațiile sale, mai exact coborârea pe gradient cu *momentum*.

## 2.2.2 Optimizare

Metoda cea mai folosită pentru modificarea ponderilor folosite de neuroni este coborârea pe gradient. Aceasta are rolul de a găsi punctul minim al unei funcții, bazându-se, după cum se prezintă în [41], pe faptul că dacă funcția pe care vrem să o minimizăm este definită și diferențiabilă în jurul unui punct, atunci valoarea aceleiai funcții descrește cel mai repede în sensul opus gradientului.

Așadar, ponderile sunt modificate în felul următor:

$$\theta = \theta - \eta * \nabla f(\theta) [31] \quad (2.2)$$

Astfel este prezentată formal ecuația 2.2 și iată ce înseamnă fiecare simbol:

- $\theta$ (theta) reprezintă un parametru(pondere)
- $\eta$ (eta) este rata de învățare, denumită uneori și  $\alpha$ (alpha)
- $\nabla$ (nabla) este gradientul, care este luat din f
- f este funcția de cost sau funcția de pierdere(*loss function*)

Metoda coborârii pe gradient are probleme și se poate bloca într-un minim local sau într-un punct "șa"(*saddle point*)(vezi Figura 2.5) care pot apărea de multe ori într-un spațiu cu multe variabile. Astfel, a fost introdusă variația coborârii pe gradient cu *momentum*[26].

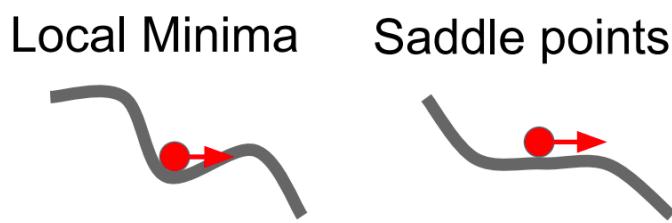


Figura 2.5: Punct de minim local și punct "șa". Figură preluată din [3]

Această metodă folosește pentru actualizarea ponderilor actuale valoarea "vitezei" sau *velocity*-ului de la pasul precedent. Ecuația 2.3 descrie modul în care funcționează coborârea pe gradient cu *momentum*, aşa cum este implementată în Pytorch, fiind ușor diferită față de formula originală, prezentă în [26] și descrisă prin Ecuația 2.4. În această lucrare de licență este folosită o valoare a *momentum*-ului de 0.9.

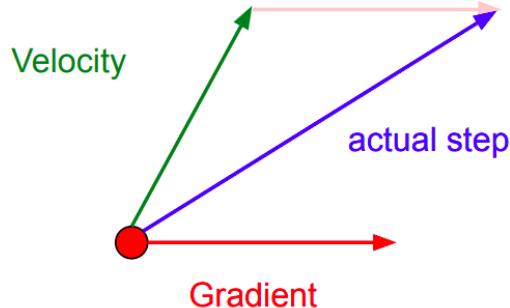


Figura 2.6: Modificarea ponderilor prin *momentum*. Figură preluată din [3]

$$\begin{aligned} v_{t+1} &= \mu * v_t + \nabla f(\theta_t) \\ \theta_{t+1} &= \theta_t - \eta * v_{t+1} \end{aligned} \quad (2.3)$$

unde

- $\theta(\theta)$  reprezintă un parametru(pondere)
- $\eta(\eta)$  este rata de învățare
- $\nabla(\nabla)$  este gradientul, care este luat din  $f$
- $\mu$  este *momentum*-ul
- $v$  reprezintă *velocity*-ul
- $f$  este funcția de cost sau funcția de pierdere(*loss function*)

$$\begin{aligned} v_{t+1} &= \mu * v_t - \nabla f(\theta_t) \\ \theta_{t+1} &= \theta_t + v_{t+1} \end{aligned} \quad (2.4)$$

## 2.3 Transfer learning

În comunitatea *deep learning* există în trecut concepția greșită care spunea că nu poți antrena modele foarte adânci cu un set de date mic(adică cu mai puțin de 1 milion de date). Însă, odată cu conceptul de *transfer learning*, acest lucru nu mai e valabil. În practică, este foarte comun să folosești modele preantrenate pe un set de date foarte mare, cum ar fi ImageNet, care conține 1,2 milioane de imagini împărțite în 1000 de clase. Această rețea îți

poate servi drept rețea de bază, *backbone*-ul rețelei convezionale pe care o folosești pentru sarcina de interes. Cele două scenarii majore sunt:

- **Rețeaua convezională ca extractor pentru caracteristici.** Se elimină ultimul strat *fully-connected* al unui ConvNet preantrenat pe ImageNet și se înlocuiește cu un nou strat *fully-connected*, cu numărul dorit de clase, iar restul de straturi rămân înghețate. Această metodă se potrivește foarte bine în momentul în care setul de date pentru noua sarcina seamănă foarte mult cu setul de date ImageNet. Această abordare a fost folosită în prezentă lucrare de licență ca experiment în antrenarea detectorului de mașini, *backbone*-ului acestuia(ResNet18) înghețându-i toate straturile și folosindu-l ca extractor de caracteristici(a fost posibil acest lucru deoarece în ImageNet există și foarte multe imagini cu mașini - detaliu în Capitolul 6).
- **Modificarea sau *finetuning*-ul ponderilor ConvNet-ului.** Această strategie presupune modificarea mai multor straturi, nu doar pe cel *fully-connected*. Este potrivită în momentul în care setul de date pentru sarcina curentă este diferit față de ImageNet, dar are și date suficiente pentru antrenare. În prezentă lucrare de licență, pentru sarcina de clasificare a mașinilor în funcție de model, am ales să fac două experimente:
  - modificarea parametrilor ultimelor două *layer*e din ResNet34(detaliu în Secțiunea 5.2) precum și a parametrilor tuturor straturilor pentru normalizarea batch-ului, deoarece media și deviația standard pentru setul de date curent sunt diferite față de cele ale ImageNet-ului. În plus, pentru SEResNet34 respectiv CBAMResNet34, modulele care se ocupă cu mecanismul de atenție au rămas de asemenea dezghețate. Această alegere este motivată de faptul că primele straturi dintr-o rețea învață de obicei caracteristici generice, cum ar fi muchiile din imagine, în timp ce straturile superioare se ocupă cu învățarea detaliilor.
  - inițializarea straturilor cu ponderile preantrenate pe ImageNet, și apoi modificarea(*finetuning*-ul) tuturor parametrilor pentru setul de date curent.

Mai multe detalii despre rezultate în Capitolul 6.

## 2.4 Hook

În Pytorch, *hook*-urile sunt funcții care pot fi atașate straturilor unui model în pasul de *forward* sau *backward*. Atunci când un strat cu *hook* atașat este apelat, acesta memorează gradientul corespunzător lui. Acest *hook* mi-a fost de mare ajutor în vizualizarea convoluțiilor în diverse straturi din ResNet34(Figura 2.3) cât și în implementarea *Grad-CAM*-ului [33](detaliu în Subsecțiunea 5.3.1) prin care îți poți da foarte ușor seama ce a determinat modelul să facă o anumită predicție.

Un exemplu de clasă pentru salvarea unui *hook* la un anumit strat dintr-un model:

Listing 2.1: Cod preluat din [25].

```
class SaveActivations():
    features = None
    def __init__(self, model, layer_num):
        self.hook = model[layer_num].register_forward_hook(self.hook_fn)
    def hook_fn(self, module, input, output):
        self.features = output.cpu()
    def remove(self):
        self.hook.remove()
```

## 2.5 Concepte necesare în evaluarea unei rețele

Precizia se calculează după Ecuația 2.5. Recall-ul este calculat folosind Ecuația 2.6. Acuratețea este calculată conform Ecuației 2.7.

$$precizie = \frac{\#predictii\_corecte\_clasa}{\#predictii\_totale\_clasa} \quad (2.5)$$

$$recall = \frac{\#predictii\_corecte\_clasa}{\#corecte\_clasa} \quad (2.6)$$

$$acuratețe = \frac{\#predictii\_corecte}{\#total} \quad (2.7)$$

Pentru a înțelege mai bine aceste concepte, voi lua un exemplu concret, conform Figurii 2.7. Astfel, precizia pentru clasa *Audi* reprezintă numărul de predicții *Audi* corecte(15)

din numărul total de predicții *Audi*( $15 + 2 = 17$ ), adică  $15 / 17 * 100 = 88.23\%$ . Pe de altă parte, recall-ul pentru *Audi* este reprezentat de numărul de predicții *Audi* corecte(15) din numărul total de exemple reale *Audi*( $15 + 5 = 20$ ), adică  $15 / 20 * 100 = 75\%$ .

Acuratețea, față de precizie și recall nu este calculată per clasă, ci per total. Aceasta este mult mai intuitivă, reprezentând numărul de predicții corecte( $15 + 21 = 37$ ) din totalul exemplelor( $15 + 5 + 2 + 21 = 43$ ), ceea ce ne duce la o acuratețe de 86%.

		Clasa prezisa	
Clasa corecta		Clasa = Audi	Clasa = BMW
	Clasa = Audi	15	5
	Clasa = BMW	2	21

Figura 2.7: Precizia și recall-ul

# Capitolul 3

## Abordări recente

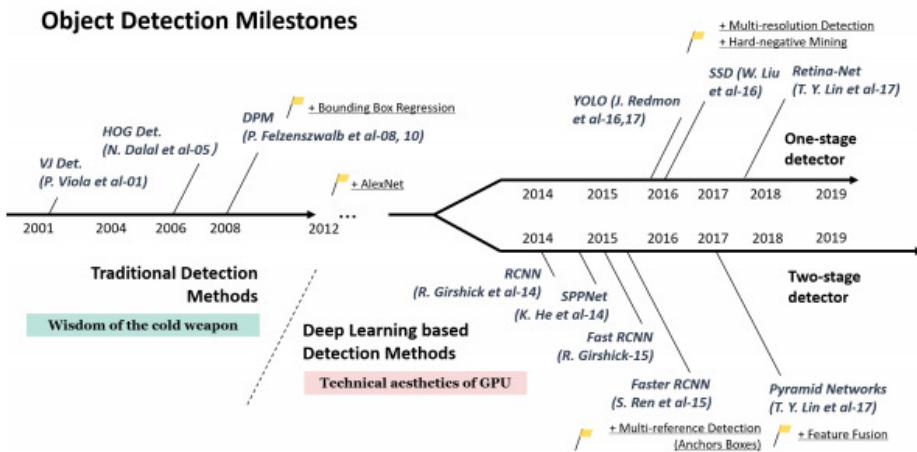


Figura 3.1: Evoluția detectoarelor de obiecte. Printre cele mai importante se numără R-CNN[12], Fast R-CNN[11], Faster R-CNN[30], YOLO[29], SSD[20]. Figură preluată din [44]

În ceea ce privește detectoarele de obiecte, s-au făcut progrese foarte mari pe planul performanței acestora, mai ales începând cu anul 2012, când AlexNet[19] a câștigat competiția ImageNet[5] și astfel a pornit era *deep learning*-ului(Figura 3.1). Când un detector este dezvoltat și prezentat publicului, acesta este antrenat pe seturi de date consacrate, cum ar fi ImageNet sau COCO(*common objects in context*), ce conțin clase generale cum ar fi vehicule, persoane, animale. Pentru setul de date ales, din căutările efectuate cu scopul realizării acestei lucrări, am ajuns la concluzia că multe soluții existente pentru rezolvarea acestei probleme implementează doar clasificarea acelor imagini în funcție de model(de exemplu, implementările disponibile la [4] sau la [17]), dar nu se ocupă și cu detecția lor. O soluție ce se apropie de prezenta lucrare de licență o reprezintă API-ul disponibil la [9]. Însă, prin prezenta

lucrare am dorit să adaptez setul de date original la contextul actual, având în vedere că multe din clasele disponibile nu se găsesc în zona noastră geografică, sau se găsesc foarte rar.

Venind după R-CNN și Fast R-CNN, Faster R-CNN a fost o inovație în domeniul deoarece a scăzut foarte mult timpul în care o inferență se realizează, prin înlocuirea algoritmului *selective search*[36], care printr-o metodă greedy combină regiunile similare până reușește să propună regiunile potrivite. Astfel, acesta ocupă majoritatea timpului necesar unei inferențe (conform [10] erau nevoie de 2.3s pentru o inferență în Fast R-CNN și de 0.3s dacă nu se folosea algoritmul pentru propunerii), reprezentând limitarea acestor detectoare. În Faster R-CNN, propunerile sunt realizate printr-o altă rețea convecțională, antrenată în același timp cu tot sistemul (detalii în Secțiunea 5.1).

# **Capitolul 4**

## **Tehnologii folosite**

### **4.1 Python**

Atât aplicația care se ocupă cu detecția și clasificarea mașinilor(aplicația de bază), cât și interfața web care comunică cu aceasta am ales să le scriu în Python[38], datorită numărului foarte mare de librării disponibile. Acesta este un limbaj interpretat și dinamic, cu o sintaxă simplă, foarte apropiată de limba engleză. Acest lucru îl face potrivit atât pentru dezvoltarea rapidă a aplicațiilor, cât și pentru mențenanța lor. Deși este un limbaj interpretat, motiv pentru care execuția codului ar dura mai mult decât în cazul limbajelor compilate, majoritatea operațiilor care necesită o putere de calcul mare se realizează în NumPy[37] și Pytorch[23].

#### **4.1.1 Aplicația de bază**

#### **4.1.2 NumPy**

Numpy[37] este o librărie de Python pentru calcule matematice complexe, ce este scrisă în mare parte în limbajul C, ceea ce o face foarte rapidă din punct de vedere computațional deoarece folosește vectorizarea în locul loop-urilor clasice care ar consuma foarte mult timp.

### 4.1.3 Pytorch

Pytorch este un framework dezvoltat de compania Facebook, destinat în principal domeniului ce se ocupă cu *deep learning*-ul, oferind foarte multă flexibilitate, dezvoltarea modelelor făcându-se într-un stil "pythonic". Datorită acestui lucru, a crescut foarte mult în popularitate de când a fost lansat publicului. În Pytorch, structura de date de bază este Tensor-ul, ce reprezintă un tablou multidimensional ce seamănă foarte mult cu matricile din Numpy, cu diferența că în Pytorch, aceste structuri pot fi mutate pe GPU, operațiile întâmplându-se astfel acolo, prin intermediul CUDA-ului.[35]

```
x = torch.empty(5, 3)      x = np.empty([5, 3])
print(x)                  print(x)
```

După cum se poate observa mai sus, creearea unei matrici goale de dimensiuni 5x3 în Numpy este aproape identică cu creearea unui Tensor în Pytorch.

### 4.1.4 OpenCV

OpenCV(*Open Source Computer Vision Library*)[2] este o librărie destinată domeniului vederii artificiale cu foarte mulți algoritmi implementați ce accelerează procesul dezvoltării unei aplicații ce presupune folosirea lor. În prezența lucrare, această librărie este folosită pentru citirea imaginilor sau pentru diverse procesări asupra lor, cum ar fi redimensionarea sau combinarea acestora(pentru crearea *heatmap*-ului obținut prin *Grad-CAM*, detalii în Subsecțiunea 5.3.1)

### 4.1.5 PIL

PIL(*Python Imaging Library*), este, după cum îi spune și numele, o librărie de Python ce se ocupă cu manipularea imaginilor. Mi-a fost utilă la încărcarea setului de date, deoarece imaginile citite cu modulul *Image* din PIL pot fi foarte ușor transformate în Tensori, astfel fiind pregătite pentru antrenament.

### 4.1.6 scikit-learn

scikit-learn[24] este folosit în domeniul *machine learning*-ului deoarece oferă soluții pentru clasificare, regresie etc. În acest proiect mi-a fost utilă funcționalitatea de creare a matricei

de confuzie pentru datele de testare, astfel putând să calculez precizia pentru diferite arhitecturi.

### 4.1.7 pandas

pandas[21] este un toolkit folosit pentru procesarea volumelor mari de informații. Acesta mi-a fost util în lucrul cu csv-uri, atunci când datele pentru antrenament / testare erau încărcate, iar fișierul foarte mare de tip csv trebuia procesat și el. pandas face acest lucru într-un timp mult mai scurt față de funcționalitatea de bază oferită de Python.

## 4.2 Interfața web

### 4.2.1 Flask

Flask[39] este un micro framework, folosit pentru dezvoltarea aplicațiilor web. S-a popularizat rapid datorită ușurinței cu care o aplicație poate fi scrisă, începând însă ca un wrapper peste Werkzeug și Jinja.

#### Dropzone Flask

Dropzone Flask[7] este folosit pentru încărcarea imaginilor din *front-end* pe server, oferind o integrare foarte ușoară și ușor de menținut.

### 4.2.2 Bootstrap

Bootstrap[22] este cel mai cunoscut framework de CSS folosit în momentul de față. Aceasta conține foarte multe şablonane ce pot fi folosite, astfel eliminând timpul pierdut cu implementarea lor. Tot *front-end*-ul se bazează pe un container de Bootstrap împărțit în *row*-uri și *div*-uri. De asemenea, altă componentă importantă pe care am folosit-o este *carousel*-ul ce permite afișarea într-un mod simplu a imaginilor dorite.

# Capitolul 5

## Abordare propusă

În elaborarea acestei lucrări de licență am luat în considerare trei metode pentru rezolvarea problemei propuse. Acestea sunt:

1. **Detector pentru modelele de mașini.** Acesta se ocupă atât de propunerea regiunilor din imagine în care ar putea exista mașini, cât și de clasificarea lor în funcție de model
2. **Detector pentru de mașini + Clasificator pentru modele.** Detectorul de mașini care să se ocupe doar de propunerea regiunilor în care se găsesc acestea și clasificarea lor în funcție de mașină/background. Ieșirea acestui detector urmând să fie intrarea unui clasificator bazat pe ResNet și antrenat special pentru modelele de mașini
3. **Detector pentru de mașini + Clasificator pentru modele cu mecanism de atenție.** Același detector de la punctul precedent. Acum, singura diferență este că ieșirea acestui detector este intrarea unui clasificator bazat pe ResNet la care am adăugat un mecanism de atenție.

În final, am ajuns la concluzia că cea mai bună abordare din punctul de vedere al rezultatelor obținute este a doua abordare, detalii suplimentare urmând să dau în continuare.

Pentru toate abordările de mai sus am folosit implementarea din Pytorch a algoritmului Faster R-CNN[30], pe care îl voi detalia în Secțiunea 5.1. Rețeaua conlovuțională prin care este trecută imaginea și din care se obțin feature map-urile necesare pentru Faster R-CNN[30] este un ResNet18 preantrenat pe setul de date ImageNet[5]. Prima abordare, fiind una de bază, a obținut niște rezultate mediocre întrucât, în opinia mea, clasificatorul din

Fast-RCNN[11] nu este suficient de puternic încât să optimizeze această problemă (clasificarea a 157 de modele de mașini). Pentru a doua, respectiv a treia soluție, abordarea a fost diferită: detectorul a fost antrenat doar pentru task-ul de a detecta mașini cu o eroare minimă. Toate aceste detecții sunt trecute printr-un algoritm de *non-maximum suppression* (NMS) cu scopul de a scăpa de acele detecții care se suprapun. După această preprocesare, detecțiile rămase sunt trecute printr-un clasificator ce are la bază ResNet34 respectiv ResNet34 cu mecanism de atenție, antrenat special pe cele 157 de clase din setul de date.

## 5.1 Faster R-CNN(RPN + Fast R-CNN)

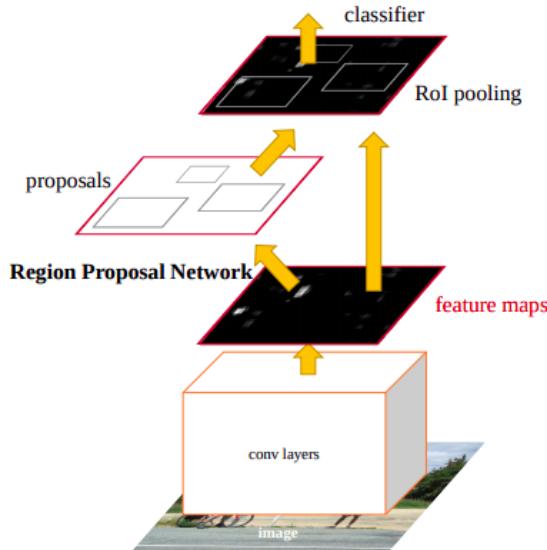


Figura 5.1: Arhitectura pentru Faster R-CNN. Figură preluată din [30]

Arhitectura Faster R-CNN[30] este folosită, în acest caz, pentru detectarea mașinilor din poze. Aceasta reprezintă combinația dintre RPN (Region Proposal Network) care propune regiuni și detectorul Fast R-CNN[11] care folosește aceste regiuni propuse pentru a da bounding box-urile finale ale mașinilor, după cum se poate observa în Figura 5.4. Astfel, Faster R-CNN este un detector în două etape. Prin utilizarea RPN-ului pentru a genera ROI-urile (*Region of Interest* - regiunile de interes), [30] nu doar că reduce timpul de propunere a regiunii de la 2s la 10ms per imagine (în R-CNN[12] și Fast R-CNN, pentru propunerea regiunilor era folosit algoritmul *selective search*[36], care creștea foarte mult timpul necesar pentru o detectie), dar și permite etapei de propunere a ROI-urilor să împartă straturile

cu următoarele etape ale detectării, ducând la o îmbunătățire generală a învățării. Figura 5.1 arată structura unificată a Fast RCNN-ului și RPN-ului.

### 5.1.1 Cum funcționează Faster R-CNN

1. Imaginea de intrare este trecută printr-un CNN(ResNet18 preantrenat pe ImageNet în acest caz) pentru a fi obținut un feature map
2. *Feature map*-ul este trecut printr-o rețea separată, numită RPN, care produce box-uri / regiuni
3. Pentru boxurile / regiunile scoase de RPN folosim mai multe straturi *fully connected* pentru a prezice clasa (clasa Mașină sau background în acest caz) + coordonatele bounding box-urilor

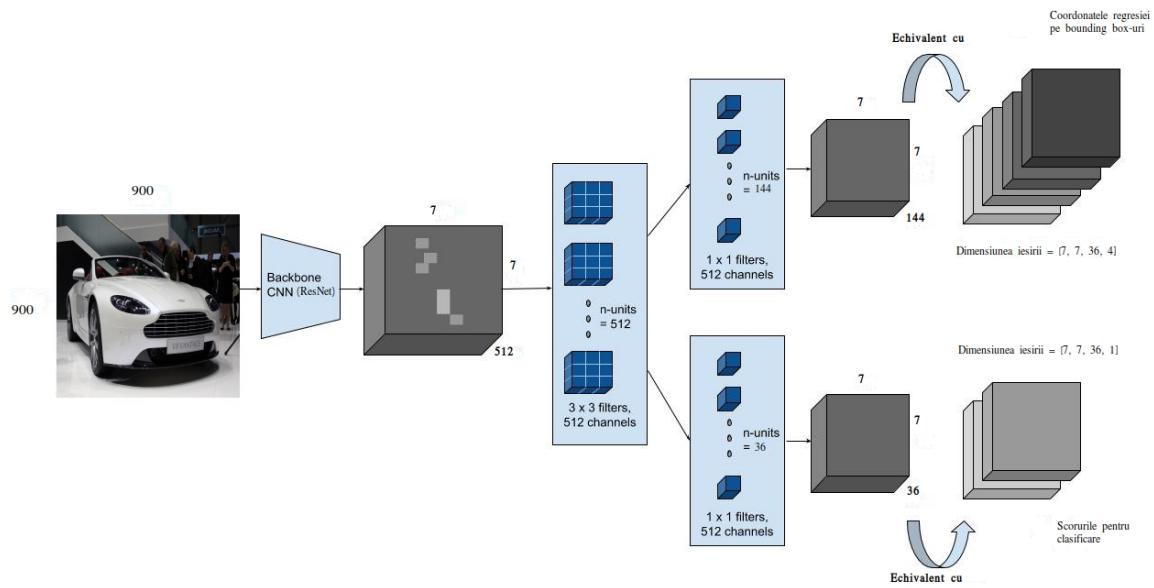


Figura 5.2: Arhitectura pentru RPN.

### 5.1.2 RPN

RPN-ul este o rețea complet conlovuțională care are rolul de a scoate în același timp bounding box-urile și scorurile pentru care un obiect se poate afla sau nu la fiecare poziție. Acest lucru se realizează prin plasarea unui set de „ancore” pe imaginea de intrare pentru fiecare locație din feature map-ul de ieșire. Aceste ancore indică posibilele obiecte(mașini) în diferite dimensiuni și raporturi de aspect în această locație. Ancorele sunt un nou mecanism, introdus prima oară în [30], folosit aici pentru să găsi mașini de diferite dimensiuni, în loc să facem convoluții pe imagini scalate în mai multe raporturi de aspect(numite piramide). Practic, mapăm fiecare locație spațială din feature map la locația sa spațială din imaginea originală, pentru ca mai apoi acestea să fie comparate cu bounding box-urile corecte și astfel să fie calculat loss-ul curent. Figura 5.3 prezintă 36 de ancore posibile în 6 raporturi de aspect diferite(1:4, 1:2, 1:1, 3:2, 2:1, 5:2) și 6 dimensiuni diferite(16, 32, 64, 128, 256, 512) plasate pe imaginea de intrare la un punct A(600, 600). Pe măsură ce rețeaua se deplasează prin fiecare pixel din harta caracteristicilor(feature map) de ieșire, trebuie să verifice dacă aceste 36 de ancore corespundătoare care acoperă imaginea de intrare conțin de fapt obiecte(mașini) și să rafineze coordonatele acestor ancore pentru a da bounding box-urile ca RoI-uri. RPN-ul operează astfel: o convoluție de 3x3 cu 512 filtre este aplicată pe harta caracteristicilor scoasă de CNN, aşa cum se poate observa în Figura 5.2, pentru a oferi o hartă de caracteristici 512-d pentru fiecare locație. Apoi, urmează două straturi paralele: un strat conlovuțional de 1x1 cu 36 de filtre pentru "objectness score"(cât de posibil este ca un obiect să se afle acolo - clasificare) și o convoluție de 1x1 cu 144(36 \* 4) de filtre pentru regresia propunerilor. O ancoră este considerată ca fiind pozitivă dacă îndeplinește oricare dintre cele două condiții:

1. ancora are cel mai mare IoU (Intersecție supra reuniune, prin care este calculată suprapunerea) cu un bounding box corect;
2. ancora are un IoU mai mare de 0.7 cu orice bounding box. Același bounding box poate face ca mai multe ancore să fie etichetate pozitiv.

O ancoră este etichetată negativ dacă IoU-ul său cu toate bounding box-urile este mai mic de 0.3. Restul ancorelor nu sunt luate în considerare.

Funcția de loss pentru RPN este o funcție cu mai multe sarcini, dată de Ecuăția 5.1.

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*) \quad [30] \quad (5.1)$$

Aici,  $i$  este indicele unei ancore într-un mini-batch, iar  $p_i$  este probabilitatea ca o ancoră să conțină un obiect.  $p_i^*$  este eticheta corectă(1 sau 0).  $t_i$  este un vector care reprezintă cele 4 coordonate ale bounding box-ului prezis, iar  $t_i^*$  este bounding box-ul adevărat. Loss-ul de clasificare  $L_{cls}$  este funcția logaritmică(*cross entropy*) pe două clase(obiect sau nu). Pentru loss-ul pe regresie se folosește funcția de loss *smooth L1*(vezi Ecuăția 5.2). Pierderea pe regresie  $L_{reg}(t_i, t_i^*)$  este activată numai dacă anora conține de fapt un obiect, adică  $p_i^*$  este 1. Cei doi termeni sunt normalizați de  $N_{cls}$  și  $N_{reg}$ , iar al doilea ponderat cu un parametru de echilibrare  $\lambda$ (în [30] autorii susțin că un lambda de 10 asigură o pondere egală între *cls* și *reg*).

$$\begin{aligned} loss\_L1\_smooth(x, y) &= \frac{1}{n} \sum_i z_i \\ &\text{unde} \\ z_i &= \begin{cases} 0.5 * (x_i - y_i), & \text{daca } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{altfel} \end{cases} \quad [11] \end{aligned} \quad (5.2)$$

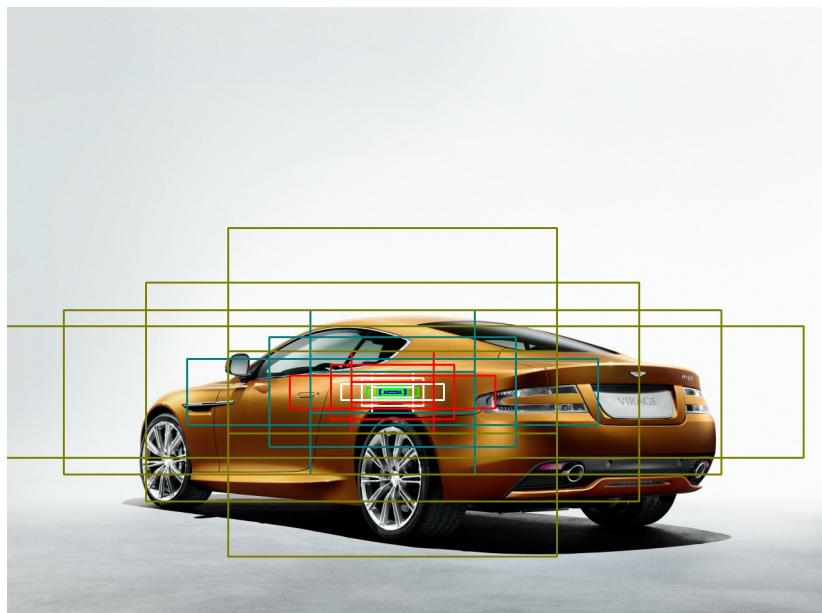


Figura 5.3: Anore la pozitia (600, 600). Fiecare dimensiune este reprezentată în altă culoare.

### 5.1.3 Fast R-CNN

Următorul pas după RPN este folosirea acestor propuneri de regiune pentru a prezice clasele obiectelor(mașinilor) și localizarea. În loc de abordarea R-CNN, unde fiecare propunere este trecută printr-o rețea de clasificare, Faster R-CNN folosește feature map-ul pentru a extrage caracteristicile. Întrucât un clasificator se așteaptă la o intrare cu dimensiuni fixe, feature map-ul este tăiat în funcție de propunerii și redimensionat la o dimensiune fixă. *Max pooling*-ul extrage cele mai importante feature-uri, ceea ce duce *region proposal*-urile la o dimensiune fixă, care sunt apoi introduse în stadiul final.

Primind aceste propuneri, acum trebuie produse probabilitățile pentru fiecare clasă și bounding box-urile finale pentru RoI(Regiunea de interes). Acest lucru se realizează folosind două straturi FC, urmate de două ramuri FC care prezic atât probabilitățile claselor dorite cât și eroarea pe bounding box-urile noastre. Stratul de clasificare produce  $N + 1$  predicții, una pentru fiecare dintre cele  $N$  clase, plus o clasă de fundal(în cazul acesta există 157 de clase). Stratul de regresie produce  $4 * (N + 1)$  predicții, unde fiecare reprezintă bounding box-ul prezis pentru fiecare dintre cele  $N + 1$  clase. Funcția de loss este asemănătoare cu cea folosită pentru RPN.

Cu toate acestea, după cum am menționat și mai devreme, propunerile de regiune vin în diferite raporturi de aspect și dimensiuni, iar datorită faptului că straturile FC primesc doar caracteristici în dimensiuni fixe, este nevoie de un strat de *RoI pooling* care va grupa toate propunerile pentru a avea aceleași dimensiuni. Înainte de toate, trebuie precizat faptul că dimensiunea imaginii de intrare în CNN-ul de bază ResNet este de aproximativ 128 de ori mai mare decât dimensiunea feature map-ului scos de acesta( $900 / 128 = 7$ ). Acest număr este important deoarece va fi folosit în cele ce urmează.

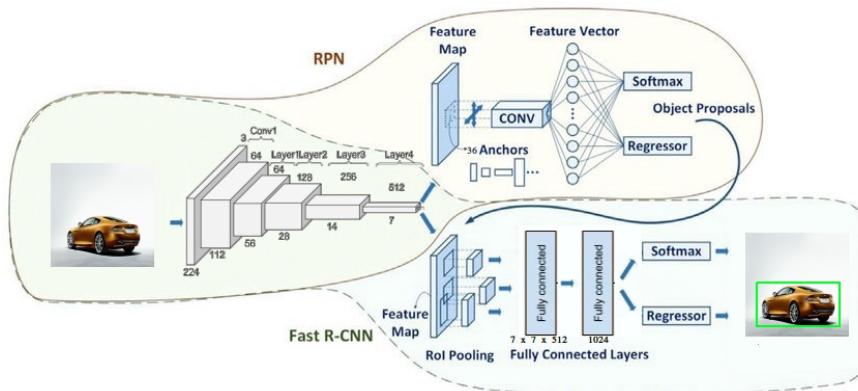


Figura 5.4: Arhitectura Faster R-CNN detaliată. Figură inspirată din [1] și [28].

Există mai multe metode pentru a redimensiona regiunile de interes, cu diferite compromisuri de pierdere a informațiilor, dar cea mai de bază este *RoI pooling*-ul care se realizează astfel:

- propunerile din spațiul imaginii de intrare se translatează în spațiul feature map-ului.

Astfel, coordonatele pot deveni numere reale, dar acestea trebuie să fie numere întregi. Apoi se rotunjesc(cuantifică) acele coordonate, provocând astfel unele pierderi de informații. Să luăm un exemplu simplu: avem o regiune de interes de dimensiune 800x640 a cărei colț stânga-sus se află în punctul de coordonate (150, 150). Pentru ca acest RoI să fie dus în spațiul hărții de caracteristici, se fac următoarele calcule:

- $x_{\min} = x_{\min} / 128;$
- $y_{\min} = y_{\min} / 128;$
- $\text{lățime} = \text{lățime} / 128;$
- $\text{înălțime} = \text{înălțime} / 128;$

Astfel, se observă că noile valori sunt  $x_{\min} = y_{\min} = 1.17$ ,  $\text{lățime} = 6.25$  și  $\text{înălțime} = 5$ . Aceste numere se rotunjesc în jos, pentru a obține niște întregi, moment în care se întâmplă prima cuantificare(*quantization*).

- RoI-ul este împărțit în funcție de dimensiunea dorită și se efectuează o operațiune de max pooling, rezultând un RoI de dimensiuni fixe care poate fi folosit ca intrare în următorul strat. Atât în paper-ul original [30] cât și în implementarea PyTorch, dimensiunea scoasă de pooling este de 7x7, dar acum vom lua drept exemplu dimensiunea de ieșire 3x3. Astfel, RoI-ul din exemplul anterior cu dimensiunea rezultată de 5x6 va fi împărțit în celule de dimensiune 1x2( $[5 / 3] = 1$ ,  $[6 / 2] = 2$ ) din care este pastrată valoarea maximă(*max pooling*). Ultimele două rânduri ale RoI-ului vor fi pierdute, având loc a două cuantificare, după cum este ilustrat și în Figura 5.5.

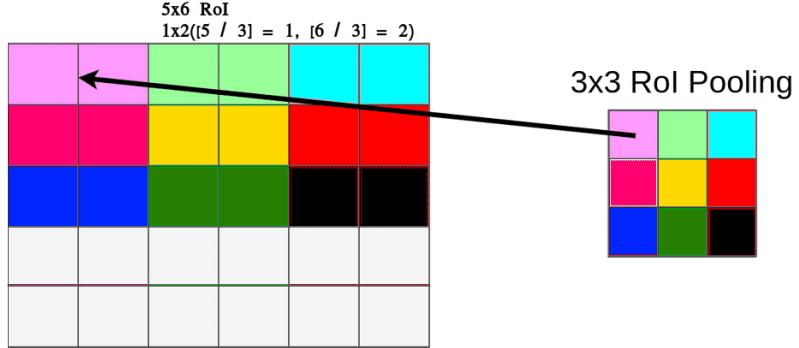


Figura 5.5: Maparea RoI-ului.

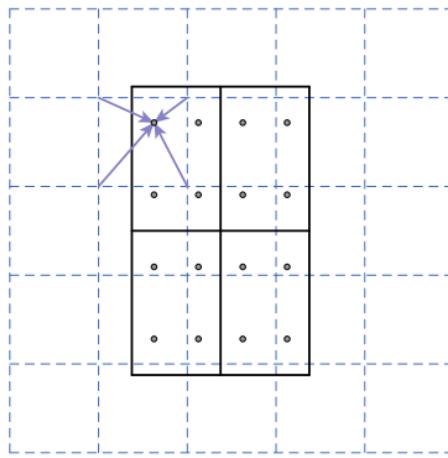


Figura 5.6: **RoIAlign**: liniile punctate reprezintă un *feature map*, liniile solide un RoI(cu celule de dimensiune 2x2) și punctele marcate reprezintă cele 4 puncte de eșantionare din fiecare celulă. RoIAlign calculează valoarea fiecărui punct de eșantionare prin interpolare biliniară folosind punctele din apropiere de pe *feature map*. Nu se efectuează cuantificări deloc. Figură preluată din [14]

Prin aceste cuantificări se pierde uneori foarte multă informație. Astfel, în implementarea de Pytorch, în loc de folosirea RoI Pooling-ului, se folosește RoI Align, introdus prima oară în paper-ul [14]. Modificarea propusă în [14] este simplă: se evită tăierea celulelor din RoI. Se folosește interpolarea biliniară(vezi Figura 5.7) pentru a calcula valorile exacte ale caracteristicilor de intrare în cele patru puncte alese în fiecare celulă RoI și rezultatul este agregat(folosind *max* sau *avg* - a se vedea Figura 5.6 pentru detalii). Folosind același RoI de mai devreme, cu lățimea de 6.25, înălțimea de 5 și punctul din stânga-sus la coordonatele (1.17, 1.17), acesta este divizat în celule cu dimensiunea de  $2.08 \times 1.66 (6.25 / 3 = 2.08, 5 / 3 = 1.66)$ . Coordonatele celor 4 puncte din fiecare celulă se calculează după formula următoare:

- $x[\text{lin}, \text{col}] = x_{\min} + (\text{lățime} / 3) * \text{col};$
- $y[\text{lin}, \text{col}] = y_{\min} + (\text{înălțime} / 3) * \text{lin}$

Astfel, primul punct are coordonatele egale cu  $x = 1.17 + (2.08 / 3) * 1 = 1.17 + 0.69 = 1.86$  și  $y = 1.17 + (1.66 / 3) * 1 = 1.72$ . La fel se procedează și pentru restul punctelor(vezi Figura 5.8). După ce toate punctele sunt calculate, se realizează interpolarea biliniară a acestora, conform Figurii 5.7. Astfel, primului punct îi corespunde valoarea  $P = 0.38(x_1 = 1.5, x_2 = 2.5, y_1 = 1.5, y_2 = 2.5$ ; vezi Figura 5.9). După ce sunt calculate valorile pentru toate cele 36 de puncte din ROI( $4 * 9$  celule), se va lua maximul(*max pooling*) din fiecare celulă și astfel vom rămâne cu un ROI de dimensiune  $3 \times 3$ .

$$P \approx \frac{y_2 - y}{y_2 - y_1} \left( \frac{x_2 - x}{x_2 - x_1} Q_{11} + \frac{x - x_1}{x_2 - x_1} Q_{21} \right) + \frac{y - y_1}{y_2 - y_1} \left( \frac{x_2 - x}{x_2 - x_1} Q_{12} + \frac{x - x_1}{x_2 - x_1} Q_{22} \right)$$

Figura 5.7: Ecuația interpolării biliniare. Figură preluată din [40]



Figura 5.8: Distribuția celor 4 puncte dintr-o celulă.



Figura 5.9: Interpolarea biliniară pentru primul punct

### 5.1.4 Antrenare

În lucrarea originală, Faster R-CNN[30] a fost antrenat folosind o abordare în mai multe etape, antrenând bucățile în mod independent și aducând împreună ponderile antrenate înapoi de a antrena rețeaua în întregime. De atunci, s-a constatat că antrenarea comună de la cap la coadă duce la rezultate similare, însă timpul de antrenare este redus cu 25-50%, conform paper-ului apărut ulterior.

După ce modelul este complet, vor exista patru funcții de loss, două pentru RPN și două pentru Fast R-CNN:

- Clasificare RPN(ancoră pozitivă / negativă)
- Regresie RPN(ancoră -> propunere)
- Clasificare Fast R-CNN(pentru clase)
- Regresie Fast R-CNN(propunere -> bounding box)

Rețeaua este antrenată folosind SGD cu momentum, cu valoarea momentum-ului de 0.9. Rata de învățare de început este 0.01, fiind împărțită cu un factor de 10 atunci când loss-ul total nu scade în maxim 5 epoci. Au fost nevoie de doar 8 epoci pentru ca loss-ul total să conveargă la 0, deoarece pentru a inițializa ResNet18 care a fost folosit pentru extragerea caracteristicilor, am folosit ponderile preantrenate pe setul de date ImageNet, care conține și imagini cu mașini. Imaginea de intrare are dimensiunea de 900x900, fiind folosite câte 8 imagini la fiecare iterație. Mai multe detalii, grafice și tabele cu rezultate se regăsesc în Capitolul 6.

## 5.2 ResNet

Modelele adânci au mai multă putere de reprezentare(mai mulți parametri) decât modelele mai puțin adânci. Astfel, cercetătorii credeau că un model mai adânc este și un model mai bun. Însă, s-a constatat că după o anumită adâncime, performanța începe să scadă, după cum se poate observa în Figura 5.10. Din această cauză, tendința generală de adăugare a mai multor straturi pentru a crea modele mai profunde s-a oprit curând din cauza unei probleme foarte frecvente în învățarea profundă care nu este cauzată de overfitting,

ci de cunoscută problemă a *vanishing gradient*-ului. Pe scurt, în timp ce un CNN se antrenează, gradientul pornește de la ultimul strat și este backpropagat prin toate straturile înainte de a ajunge la straturile inițiale. Acest lucru poate face gradientul să tindă la 0 din cauza înmulțirilor repetate, ceea ce face dificilă antrenarea straturilor inițiale ale modelului, facându-l să rămână blocat în același punct. Cu alte cuvinte, rețelele adânci sunt mult mai greu de optimizat. De aceea pentru VGG[34] nu au putut să mai adauge straturi, deoarece ar fi pierdut capacitatea de generalizare (adâncimea maximă pentru VGG este de 19 straturi). Conexiunile reziduale s-au popularizat odată cu introducerea modelelor ResNet care au creat drumuri alternative pe care gradientul poate să sară și să ajungă direct la straturile inițiale. Lucrul acesta le-a permis autorilor să antreneze modele extrem de profunde, care nu ar fi fost performante înainte (adâncimea maximă pentru ResNet este de 152 de straturi).



Figura 5.10: Arhitectura cu 56 de straturi are performanțe mai proaste atât pe antrenare cât și pe testare față de arhitectura cu 20 de straturi. Figură preluată din [3]

Astfel, în [13] autorii susțin că putem folosi straturile rețelei pentru a învăță o mapare reziduală în locul mapării dorite. Formal, dacă avem maparea de bază  $\mathcal{H}(x)$ , unde  $x$  reprezintă input-ul primului *layer*, lăsăm straturile să mapeze o altă funcție  $\mathcal{F}(x) := \mathcal{H}(x) - x$ . Maparea originală se transformă în  $\mathcal{F}(x) + x$  (Figura 5.11). Aceștia susțin că optimizarea acestei mapări reziduale este mult mai ușoară decât optimizarea mapării de bază dorite.

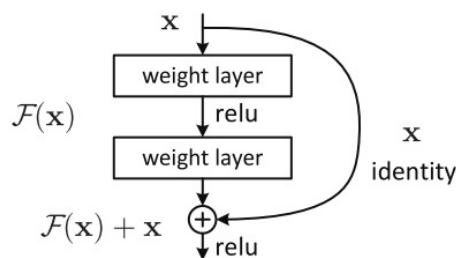


Figura 5.11: Bloc rezidual. Figură preluată din [13]

După cum se poate observa în Figura 5.13, o rețea ResNet constă din blocuri reziduale legate unul de altul. Toate blocurile au straturi convoluționale cu filtru de dimensiunea

$3 \times 3$ . Exceptie fac insă modelele mai adânci și anume Resnet-50/101/152 care folosesc un *bottleneck* ce se poate observa în Figura 5.12, partea dreaptă. Acest tip de bloc ajută la îmbunătățirea eficienței folosind convoluții cu filtru de dimensiune  $1 \times 1$  în modul următor (valabil pentru  $\text{conv2\_x}$  - ResNet50 din Tabelul 5.1, însă operațiile sunt analog pentru restul  $\text{conv}_i\text{x}$ ) - ResNet50:

- o convoluție de  $1 \times 1$ , 64 de filtre pentru a projecța  $D \times D \times 64$ , unde  $D$  reprezintă dimensiunea spațială a input-ului
- o convoluție de  $3 \times 3$  operează doar pe 64 de feature map-uri
- ultima convoluție de  $1 \times 1$ , 256 de filtre, ne proiectează înapoi în spațiul inițial al input-ului  $D \times D \times 256$

Înainte de a merge mai departe, vom stabili niște convenții. Conform Tabelei 5.1, o rețea ResNet este întotdeauna formată dintr-un strat convoluțional cu filtru  $7 \times 7$ , un strat pentru normalizarea batch-ului, o funcție de activare ReLU și un strat de *max pooling*. Toate acestea sunt urmate de patru "straturi", indiferent de variația ResNet-ului. Pentru a face rețeaua mai adâncă, pur și simplu numărul de blocuri reziduale din interiorul acestor "straturi" este crescut. În continuare mă voi referi la acest "strat" prin cuvântul *layer*, pentru a face diferența dintre un simplu strat, convoluțional de exemplu, și *layer* care se referă la concatenarea mai multor blocuri reziduale.

Liniile punctate reprezintă modificarea dimensiunii spațiale a input-ului, și anume înjumătățirea acesteia, dar dublarea numărului de filtre. Această reducere este realizată prin mărirea stride-ului de la 1 la 2 pentru prima convoluție din *layerele 2, 3, respectiv 4*.

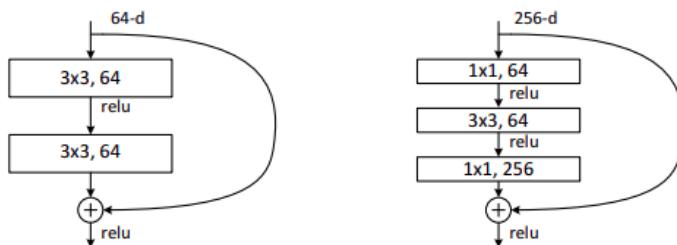


Figura 5.12: O funcție reziduală mai adâncă  $\mathcal{F}$  pentru ImageNet. Stânga: un bloc de bază sau *BasicBlock* (pe feature map-uri de  $56 \times 56$ ) ca în Figura 5.13 pentru ResNet34. Dreapta: un *Bottleneck* pentru ResNet-50/101/152. Figură preluată din [13]

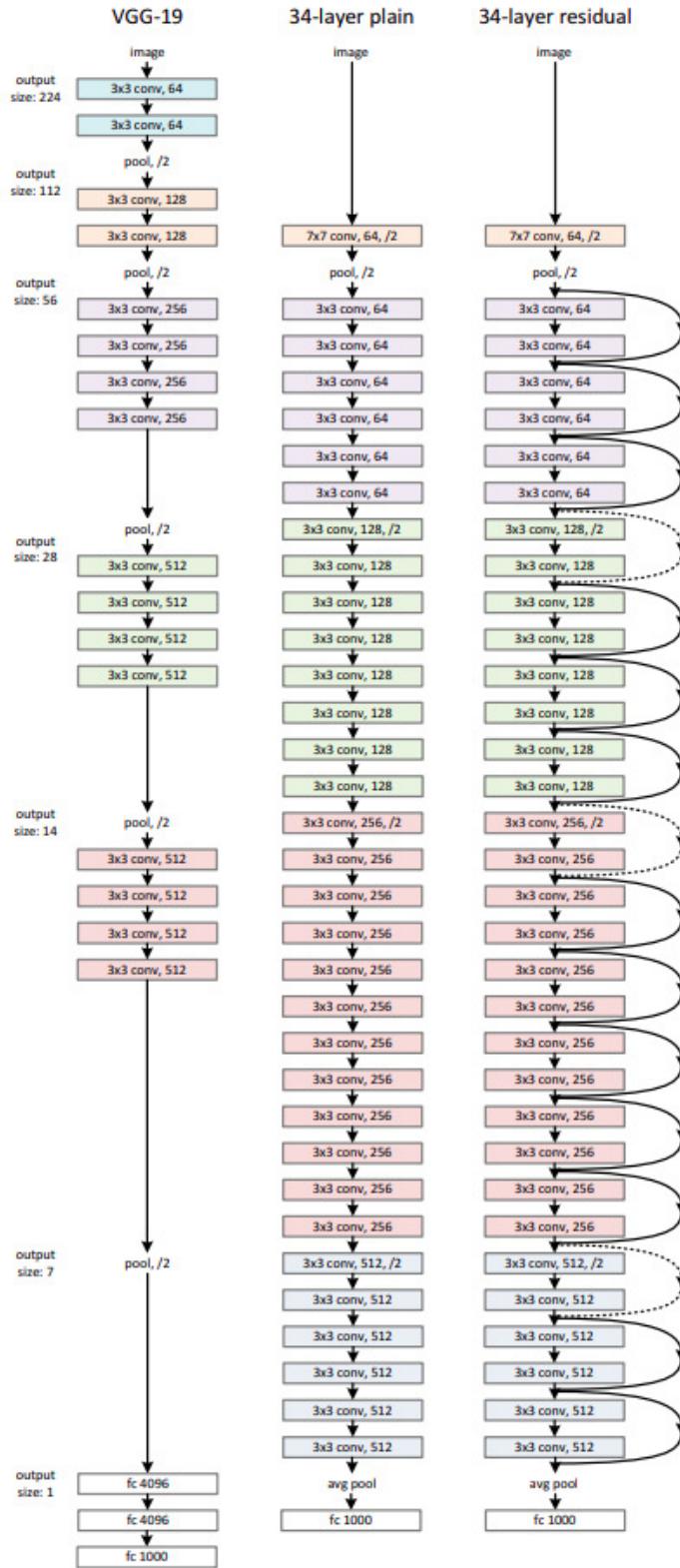


Figura 5.13: Exemple de arhitecturi pentru ImageNet. Stânga: Modelul VGG-19 [34]. Mijloc: o rețea simplă cu 34 de straturi. Dreapta: o rețea reziduală cu 34 de straturi. Tabelul 5.1 prezintă mai multe detalii și alte variante. Figură preluată din [13]

### 5.2.1 Prima conoluție

Înainte de a intra în *layerele* cu blocuri reziduale, ResNet are un "bloc", denumit în Tabelul 5.1 conv1. Acesta este format dintr-o conoluție, o normalizare a batch-ului, o funcție de activare ReLU și o operație de *max pooling*.

În Figura 5.14 se poate observa că pentru conoluție se folosesc 64 de filtre de dimensiune 7x7, cu un pas de 2. Intrarea este bordată (*padded*) cu 3 pixeli pe fiecare dimensiune spațială. De asemenea, se poate vedea că dimensiunea ieșirii este 112x112. Deoarece fiecare filtru scoate un volum pe câte un canal, volumul de ieșire ajunge să aibă dimensiunea 112x112x64.

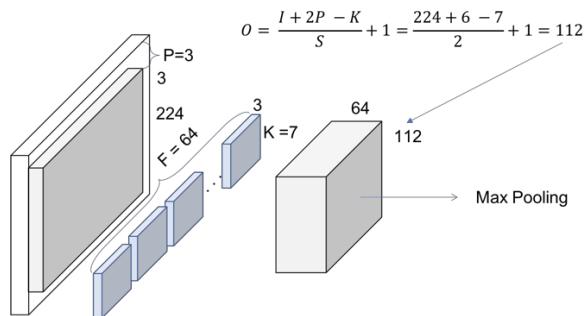


Figura 5.14: conv1 - Stratul de conoluție. Figură preluată din [1]

Următorul pas este normalizarea batch-ului, care nu schimbă dimensiunea intrării. În cele din urmă, avem funcția de activare ReLU, urmată de operația de *max pooling* cu un filtru de 3x3 și un pas de 2. Pentru ca ieșirea să aibă dimensiunea dorită, intrarea este din nou bordată cu 1 pixel pe fiecare dimensiune spațială, fiind utilizat un pas cu valoarea 1.

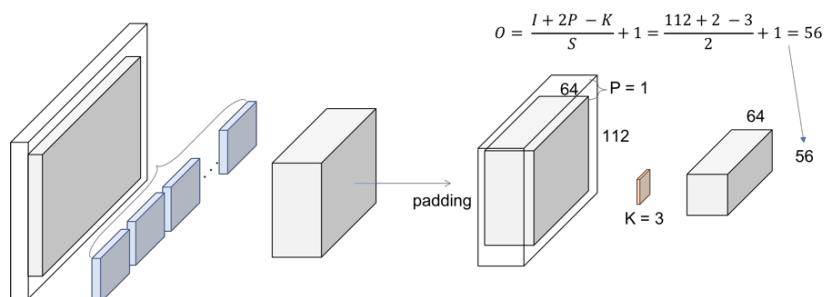


Figura 5.15: conv1 - *max pooling*. Figură preluată din [1]

## 5.2.2 Layerele din ResNet

Am precizat mai devreme faptul că un *layer* este format din mai multe blocuri, care la rândul lor sunt formate din două operații(pentru blocurile de bază, folosite în ResNet18 și ResNet34), respectiv trei operații(pentru blocurile *bottleneck*, folosite în ResNet50, ResNet101 și ResNet152). Cum în această lucrare de licență am folosit doar ResNet18, respectiv 34, mă voi referi în continuare doar la blocurile reziduale de bază. O operație se referă la o conoluție, o normalizare a batch-ului și o activare ReLU, cu excepția ultimei operații a unui bloc, care nu folosește ReLU. Figura 5.16 ilustrează mai bine ce se întâmplă în interiorul primei operații a primului bloc.

### Primul bloc

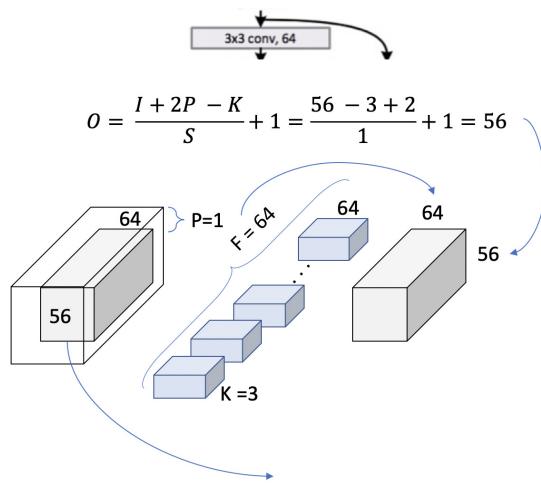


Figura 5.16: Primul *layer*, primul bloc, prima operație. Figură preluată din [1]

Conform paper-ului original, sunt folosite 64 de filtre cu dimensiunea 3x3, iar dimensiunea de ieșire este 56x56. De asemenea, dimensiunea volumului nu se modifică într-un bloc. Acest lucru este datorat faptului că se folosește o bordare cu 1 și un pas de 1. Figura 5.17 arată cum acest lucru este extins la tot blocul, și astfel ajungem la două operații per bloc, aşa cum se poate vedea în Tabelul 5.1.

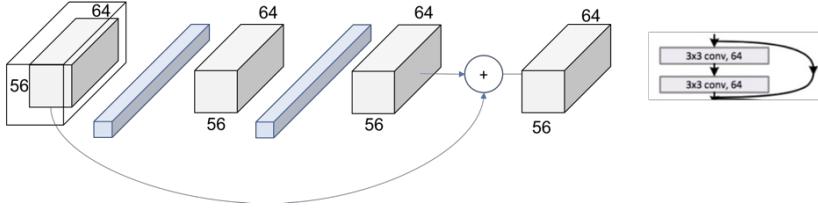


Figura 5.17: Primul *layer*, primul bloc. Figură preluată din [1]

Aceeași procedură poate fi extinsă la întregul *layer*, ca în Figura 5.18. Acum, am explicitat întreaga celulă  $\text{conv}_x$  a tabelului lui 5.1 (*primullayer – ul din ResNet34*).

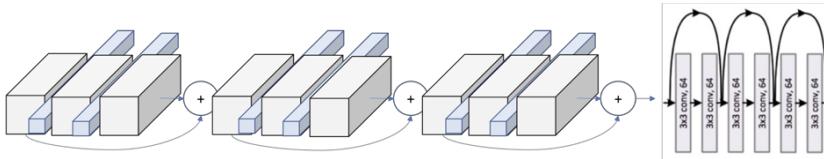


Figura 5.18: Primul *layer*. Figură preluată din [1]

### 5.2.3 Tipare în construcția ResNet-ului

În continuare voi prezenta legătura dintre *layer-urile*. În Figura 5.13 putem vedea cum acestea sunt de diferite culori. Prima operație din *layer-ele 2, 3, respectiv 4* este realizată cu pasul 2, în loc de 1 cum se întâmplă în rest, fapt marcat în Figura 5.13 prin linii punctate, reprezentând reducerea dimensionalității spațiale, și creșterea dimensiunii pe canale. Asta înseamnă că micșorarea volumului este realizată prin creșterea pasului în loc de folosirea operației de *pooling*, aşa cum se obișnuiește în rețelele convolutionale. În ResNet este folosit un singur *max pooling*, înainte de începerea *layerelor* și o operație de *average pooling*, după acestea. De aceea trebuie redimensionat și volumul care trece prin conexiunea identitate, astfel încât să le putem aduna aşa cum este reprezentat în Figura 5.17.

În paper-ul [13], această diferență este denumită *identity shortcut* și *projection shortcut*.

*Identity shortcut*-ul se referă la folosirea intrării aşa cum este, fară a fi aplicată vreo covoluție asupra ei. *Projection shortcut*-ul are grijă ca aceste două volume să se poată aduna, deci să aibă aceeași dimensiune. Autorii paper-ului investighează mai multe variante, printre care utilizarea unei covoluții de  $1 \times 1$  cu un pas de 2, care a dat cele mai bune rezultate pentru ei, și care este utilizată și în implementarea PyTorch.

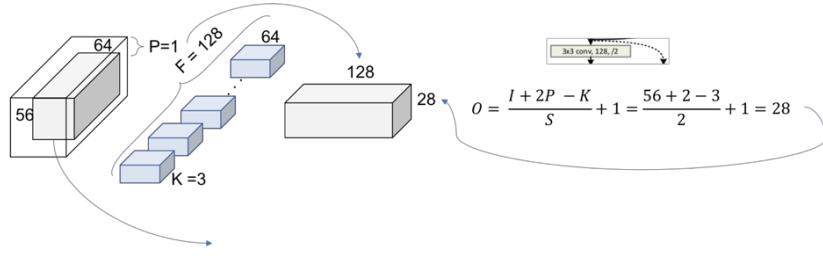


Figura 5.19: Al doilea *layer*, primul bloc, prima operație. Figură preluată din [1]

În Figura 5.19 este reprezentată această reducere a dimensiunii, prin creșterea pasului la 2. Numărul de filtre este dublat pentru a pastra complexitatea fiecărei operații( $56 * 64 = 28 * 128$ ). În momentul acesta, adunarea dintre ieșirea acestor operații și ramura cu identitatea nu poate fi efectuată deoarece dimensiunile nu sunt aceleasi. În identitate, se aplică procedura prezentată mai sus, și anume convoluția de  $1 \times 1$ (vezi Figura 5.20).

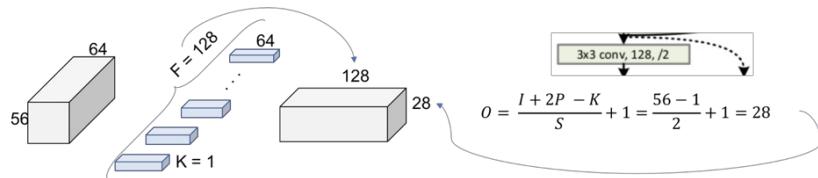


Figura 5.20: *Projection Shortcut*. Figură preluată din [1]

Figura 5.21 arată cum cele 2 volume de ieșire ale fiecărei ramuri au aceeași dimensiune și pot fi adunate.

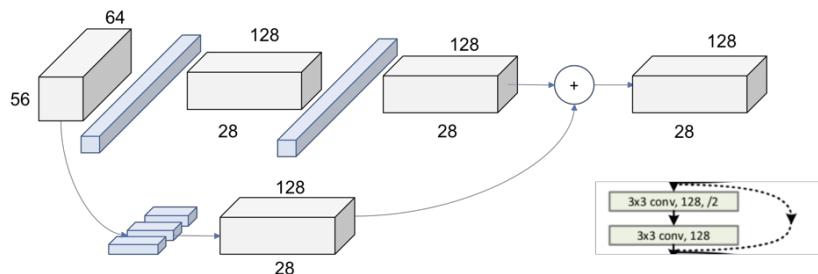


Figura 5.21: Al doilea *layer*, primul bloc. Figură preluată din [1]

În Figura 5.22 se poate vedea imaginea de ansamblu a *layer*-ului al doilea. Aceleași operații se întâmplă și pentru următoarele două *layer*e, schimbându-se doar dimensiunile volumelor primite.

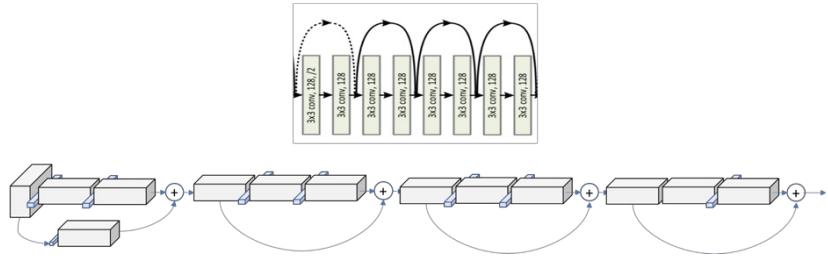


Figura 5.22: Al doilea *layer*. Figură preluată din [1]

nume_strat	dimensiune_output	resnet18	resnet34	resnet50
conv1	112x112		7x7, 64, stride 2	
conv2_x	56x56		3x3 max pool, stride 2	
		[3x3, 64] [3x3, 64]	[3x3, 64] [3x3, 64]	[1x1, 64] [3x3, 64] x 3
		x2	x3	[1x1, 256]
conv3_x	28x28	[3x3, 128] [3x3, 128]	[3x3, 128] [3x3, 128]	[1x1, 128] [3x3, 128] x 3 [1x1, 512]
conv4_x	14x14	[3x3, 256] [3x3, 256]	[3x3, 256] [3x3, 256]	[1x1, 256] [3x3, 256] x 3 [1x1, 1024]
conv5_x	7x7	[3x3, 512] [3x3, 512]	[3x3, 512] [3x3, 512]	[1x1, 512] [3x3, 512] x 3 [1x1, 2048]
	1x1	1x1 average pool, 157-d fc, softmax		

Tabela 5.1: Arhitecturi ResNet pentru ImageNet. Blocurile de construcție sunt prezente între paranteze (vezi și Figura 5.12), cu numărul de blocuri stivuite. Downsampling-ul este realizat de conv3\_1, conv4\_1 și conv5\_1 cu un pas(stride) de 2. Tabel inspirat din [13]

Funcția de *loss* pentru clasificatorul responsabil cu modelele mașinilor este o funcție clasică și anume *cross entropy*, reprezentată de Ecuăția 5.3:

$$\text{loss}(y, \text{class}) = -\log\left(\frac{e^{y[\text{class}]}}{\sum_j e^{y[j]}}\right) = -y[\text{class}] + \log\left(\sum_j e^{y[j]}\right) [23] \quad (5.3)$$

unde:

- $y$  reprezintă probabilitățile calculate pentru fiecare clasă după formula  $\mathbf{W} * \mathbf{x}$ ,  $\mathbf{W}$  reprezentând ponderile și  $\mathbf{x}$  input-urile

- class reprezintă clasa corectă

### 5.2.4 Mecanism de atenție

Autorii lucrărilor [15] [42] nu au încercat să creeze o nouă arhitectură care să dea rezultate mai bune în competițiile cunoscute, ci au încercat să le facă mai bune pe cele existente creând niște module ce se pot integra foarte ușor. Ideea de atenție într-un model neuronal a apărut prima oară în domeniul *NLP*-ului, fiind apoi folosit și în domeniul vederii artificiale. Scopul principal al acestor mecanisme este concentrarea pe caracteristicile importante și scăparea de cele care nu sunt necesare.

#### ***SE(Squeeze and Excitation)***

În [15] autorii susțin că ieșirea produsă de o conoluție amestecă dependințele specifice canalelor cu corelația spațială produsă de acele filtre. Din această cauză, ei își doresc să ofere acces global filtrelor, recalibrându-le în două faze: *squeeze* și *excitation*.

În faza de *squeeze*, o operație de *global avg pooling*(*avg pooling* de-a lungul canalelor) este aplicată întrării, încorporând astfel informația globală a *feature map*-urilor. Acest lucru este motivat de faptul că o regiune din blocul  $U$  (Figura 5.23) vede doar informația locală, fără a avea acces la contextul global al imaginii.

În faza de *excitation*, dependințele de pe axa canalelor sunt capturate folosind un simplu *MLP(multi layer perceptron)*, format din două straturi *fully-connected*, cu funcția de activare ReLU aplicată pe primul, peste care este aplicată funcția de activare sigmoid. În final, ieșirea acestui bloc este redimensionată la dimensiunile normale prin înmulțirea de-a lungul canalelor cu volumul inițial  $U$ .

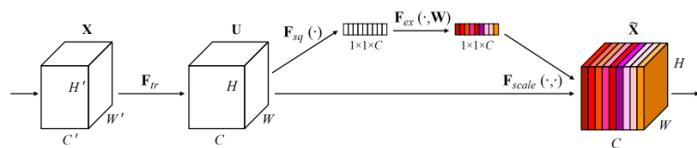


Figura 5.23: Un bloc *Squeeze and excitation*. Figură preluată din [15]

Tot acest modul este apoi integrat într-un bloc rezidual de bază(folosit în această lucrare pentru SEResNet34), conform Figurii 5.24.

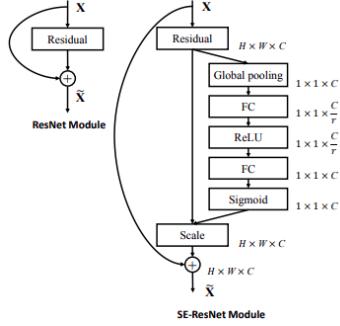


Figura 5.24: Stânga: un bloc rezidual. Dreapta: un modul SEResNet folosit și în prezență lucrare de licență. Figură preluată din [15]

### **CBAM(*Convolutional Block Attention Module*)**

Într-o oarecare opoziție cu ce spun autorii în [15], autorii [42] susțin că atenția merită concentrată atât pe axa canalelor, cât și pe dimensiunea spațială a intrării. Astfel, aceștia combină un modul aplicat pe axa canalelor intrării, care învăță "ce" caracteristici sunt importante și un modul aplicat pe dimensiunea spațială, care învăță "unde" găsește acele caracte-ristici, conform Figurii 5.25.

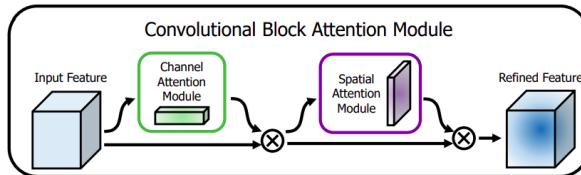


Figura 5.25: Prezentare generală a CBAM-ului. Modulul are două sub-module secvențiale: pe adâncimea canalelor și pe dimensiunea spațială. Figură preluată din [42]

Modulul care se ocupă cu atenția pe canale este identic cu modulul SE prezentat mai de-vreme, cu singura diferență că pe intrare, pe lângă *global avg pooling*, se aplică și un *global max pooling*, cele două ieșiri fiind introduse în același *MLP*, iar rezultatele sunt adunate și funcția sigmoid este aplicată, urmând ca acest rezultat să fie înmulțit *element-wise* cu F, rezultând volumul  $F'$ (Figura 5.26). Pentru modulul spațial, peste intrare este aplicat atât *max pooling* cât și *avg pooling*, cele două rezultate fiind concatenate și trimise unui strat convolutional cu dimensiunea 7x7. Rezultatului îi este aplicat sigmoid-ul, apoi înmulțit *element-wise* cu  $F'$ , din care rezultă volumul  $F''$ (Figura 5.26).

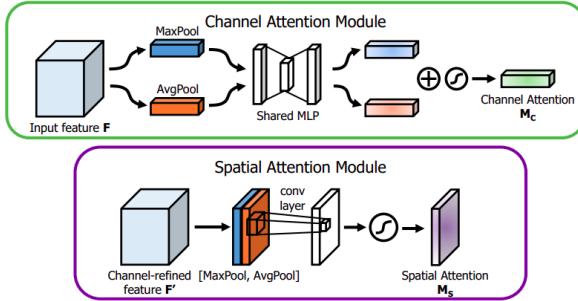


Figura 5.26: Diagrama fiecărui submodul de atenție. Submodulul pentru canale folosește atât ieșiri de *max-pooling*, cât și ieșiri de *avg-pooling* cu o rețea partajată; submodulul spațial utilizează două ieșiri similare, care sunt concatenate de-a lungul axei canalelor și introduse într-un strat convoluțional. Figură preluată din [42]

Tot acest modul este apoi integrat într-un bloc rezidual de bază(folosit în această lucrare pentru CBAMResNet34), conform Figurii 5.27.

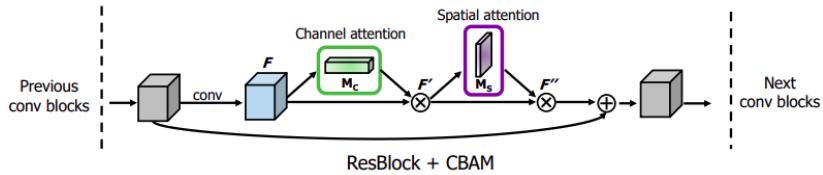


Figura 5.27: CBAM integrat cu un bloc rezidual în ResNet[13]. Figură preluată din [42]

## 5.2.5 Antrenare

Rețeaua este antrenată folosind SGD cu momentum, cu valoarea *momentum*-ului de 0.9.

Rata de învățare de început este 0.01, fiind împărțită cu un factor de 10 atunci când loss-ul total nu scade în maxim 5 epoci. Rețeaua a fost antrenată timp de 50 de epoci, fiind inițializată cu ponderile calculate pe ImageNet. Dimensiunea imaginii de intrare este 224x224, aceasta fiind dimensiunea standard pentru [5]. Dimensiunea batch-ului la antrenare este de 64 de imagini. Mai multe detalii, grafice și tabele cu rezultate se regăsesc în Capitolul 6.

## 5.3 Evaluarea metodei

Evaluarea se face folosind media preciziilor medii (*mAP - mean average precision*) la un prag al IoU-ului de 0.5. *mAP*-ul este o metrică folosită în mod obișnuit pentru evaluarea problemelor de detectare a obiectelor. *mAP*-ul penalizează atunci când este ratat un bounding box care ar fi trebuit detectat, precum și când este detectat ceva care nu există sau este

detectat același lucru de mai multe ori.

Precizia medie pentru fiecare clasă în parte este calculată folosind aria de sub curba Precizie x Recall. Deoarece de multe ori această curbă are forma unui zigzag(descreșterea nu este netedă), competiția VOC Pascal [8] folosește precizia interpolată(din 2010, interpolată în toate punctele graficului). Aceasta este interpolată în toate cele  $n$  puncte în felul următor:

$$\sum_{n=0} (r_{n+1} - r_n) \rho_{interp}(r_{n+1})$$

cu

$$\rho_{interp}(r_{n+1}) = \max_{\tilde{r}: \tilde{r} \geq r_{n+1}} \rho(\tilde{r})$$

unde  $\rho(\tilde{r})$  este precizia calculată la recall-ul  $\tilde{r}$ .

*Flow*-ul pentru testarea metodei este asemănător cu folosirea aplicației în practică: fiecare imagine este trecută prin detectorul de mașini care scoate toate detectiile; toate aceste detectii sunt apoi trecute prin clasificatorul antrenat pe modelele de mașini și astfel mașina este detectată și clasificată.

Pentru a evalua detectorul, am folosit codul pus la dispoziție de [27].

### 5.3.1 *Grad-CAM*

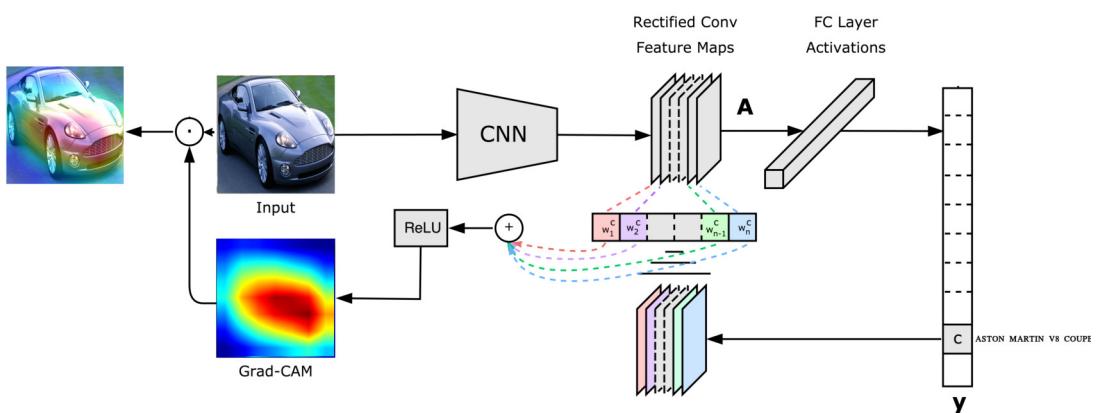


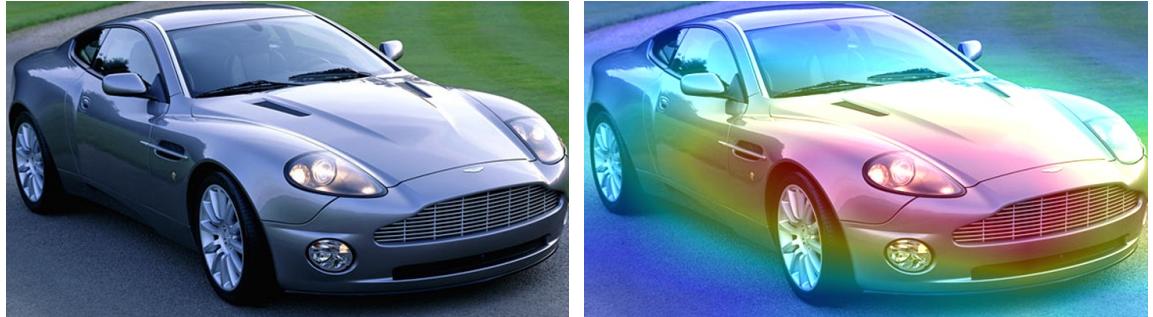
Figura 5.28: Modul de funcționare al *Grad-CAM*-ului. Figură inspirată din [33]

Pe lângă clasică testare în funcție de metrică, am ales să și vizualizez regiunile din imagine care au avut un impact major în clasificarea acesteia într-o anumită clasă. Tehnica pe

care am folosit-o pentru a îndeplini această sarcină poartă denumirea de *Grad-CAM*[33], prescurtare de la *Gradient weighted Class Activation Map*. Față de precedenta metodă, *CAM*[43], care presupunea modificarea rețelei antrenate și înlocuirea straturilor *fully-connected* cu un strat de *global average pooling*, pentru [33] nu mai este nevoie de modificarea arhitecturii, putând fi folosit în orice tip de arhitectură. Pentru a obține harta de localizare specifică clasei, Grad-CAM calculează gradientul  $y^c$  (scorul pentru clasa c) ținând cont de *feature map*-urile A ale unui strat convolutional. Pe acești gradienți este aplicat un *global average pooling* cu scopul de a obține ponderile de importanță  $\alpha_k^c$ (Ecuăția 5.4). Similar cu CAM, *heatmap*-ul(Figura 5.29) generat de Grad-CAM este o combinație ponderată de *feature map*-uri, peste care este aplicat ReLU(Ecuăția 5.5).

$$\alpha_k^c = \underbrace{\frac{1}{Z} \sum_i \sum_j}_{\text{gradients via backprop}} \underbrace{\frac{\partial y^c}{\partial A_{ij}^k}}_{\text{global average pooling}} \quad [33] \quad (5.4)$$

$$L_{Grad-CAM}^c = \text{ReLU}\left(\sum_k \alpha_k^c A^k\right) \quad [33] \quad (5.5)$$



(a) Exemplu din setul de testare

(b) *Heatmap*-ul exemplului din setul de testare

Figura 5.29: *Heatmap* generat de Grad-CAM

# Capitolul 6

## Experimente și rezultate

### 6.1 Pregătirea datelor

Setul de date[18] ce stă la baza celui folosit de mine pentru acest task conține 16.185 de imagini împărțite în 196 de clase de mașini. Fișierul cu adnotări puse la dispoziție este în format Matlab, ceea ce înseamnă că eu a trebuit să îl convertesc în format csv(*comma-separated values*) pentru a putea lucra cu el. Fișierul astfel rezultat are forma *[filename, width, height, class\_id, xmin, ymin, xmax, ymax]*, unde:

- *filename* reprezintă numele imaginii
- *width* reprezintă lățimea imaginii
- *height* reprezintă înălțimea imaginii
- *class\_id* reprezintă id-ul corespunzător clasei căreia îi aparține imaginea respectivă
- $(xmin, ymin)$  reprezintă coordonatele celui mai din stânga-sus punct al bounding-box-ului după cum se poate observa în Figura 6.1
- $(xmax, ymax)$  reprezintă coordonatele celui mai din dreapta-jos punct al bounding-box-ului după cum se poate observa în Figura 6.1

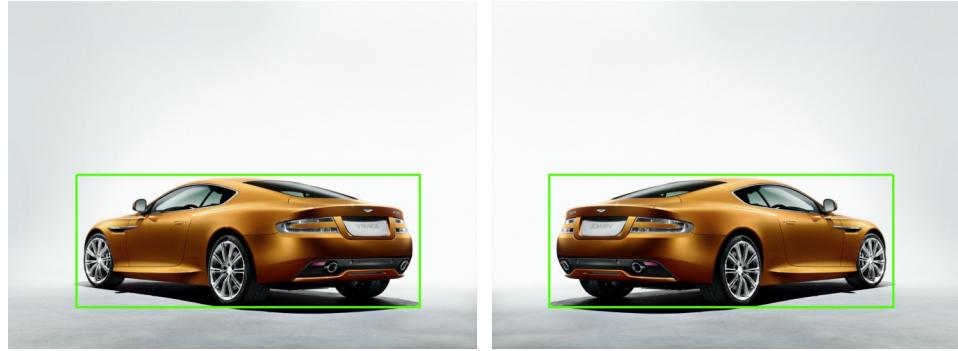


Figura 6.1: Reprezentarea bounding-box-urilor

Însă, eu nu am folosit acest set de date ca atare ci l-am modelat pentru a se potrivi mai bine contextului în care mă aflu. În acest sens, am abordat curățarea setului de date în două feluri:

- Din perspectiva brand-ului: în acest set de date se află multe modele de mașini(Acura, Buick, Daewoo, Geo Metro, GMC, Isuzu, Lincoln, Plymouth, RAM, Scion, Spyker, Eagle, Fisker, Hummer) care se găsesc exclusiv în SUA sau care se găsesc foarte rar în Europa. Astfel, am considerat utilă eliminarea acestor brand-uri
- Din perspectiva anului de fabricație a mașinii: am considerat utilă și eliminarea modelelor de mașini mai vechi de anul 2000

După toate aceste procesări, am rămas cu un total de 12.834 de imagini și 157 de clase. Pentru antrenarea detectorului de mașini, sunt folosite 70% din toate aceste imagini, însemnând 8.983 de date pentru antrenament. În stadiul acesta, neavând nevoie de id-ul clasei pentru clasificarea în funcție de model, toate bounding box-urile din fișierul csv primesc eticheta 1, corespunzătoare clasei *Car*. Pentru a evita *overfitting*-ul și pentru a adăuga exemple de antrenare, am ales să aumentez imaginile prin întoarcerea lor pe orizontală(vezi Figura 6.2), cu o probabilitate de 50%. Toate imaginile sunt redimensionate la dimensiunea spațială de 900x900 de pixeli și normalizate cu media și deviația standard calculate pe setul de date ImageNet( $mean\_nums = [0.485, 0.456, 0.406]$ ,  $std\_nums = [0.229, 0.224, 0.225]$ , fiecare valoare fiind corespunzătoare unuia din cele 3 canale).



(a) Exemplu din setul de antrenare      (b) Exemplu augmentat din setul de antrenare

Figura 6.2: Augmentare pentru detector.

Pentru a lucra cu imaginile în problema clasificării în funcție de model, imaginile disponibile în setul original [18] au fost tăiate conform bounding box-urilor disponibile în fișierul csv, pentru a se pune accent pe detaliile mașinii prezente în poză și nu pe fundal. Acestea au fost împărțite în directoarele corespunzătoare fiecărei clase, iar de aici clasa *ImageFolder* disponibilă în PyTorch s-a ocupat de încărcarea datelor și atribuirea etichetei corespunzătoare. Am ales să folosec 80% din totalul imaginilor disponibile, dintre care 10% pentru validarea modelului în timpul antrenamentului. Ca și în cazul detectorului, pentru a adăuga mai multe exemple de antrenare(sunt în jur de 50-70 de imagini disponibile per clasă), datele au fost augmentate prin diverse transformări implementate în PyTorch. Printre acestea se numără întoarcerea pe orizontală cu o probabilitate de 50%, rotația aleatoare a imaginii cu un unghi de 15 grade, cât și *color jitter*-ul care schimbă în mod aleator contrastul și luminozitatea imaginii. Toate imaginile sunt redimensionate la dimensiunea de 224x224 pixeli înainte de a li se aplica toate aceste transformări, iar la final sunt normalizate în același mod în care sunt și cele pentru antrenarea detectorului. Figura 6.3 ilustrează cum arată toate aceste preprocesări asupra imaginii.

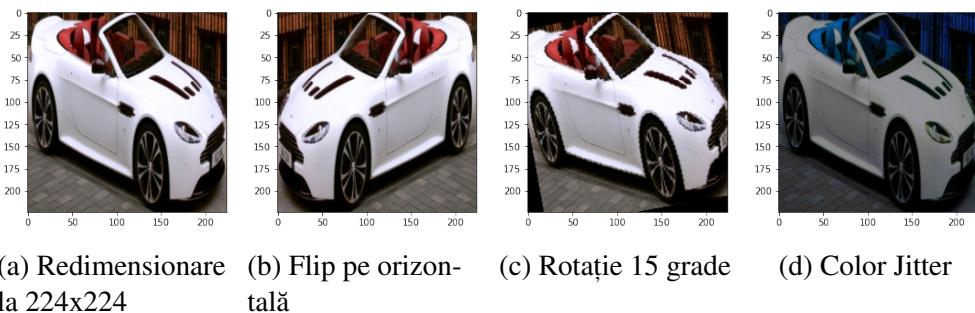


Figura 6.3: Diverse transformări

## 6.2 Detecția de mașini

Pasul pentru detecția mașinilor joacă un rol foarte important în predicția finală scoasă de arhitectură, deoarece dacă detectorul nu vede toate mașinile, acestea nu pot fi clasificate de către clasificatorul antrenat pentru modelele mașinilor. Astfel, am făcut mai multe teste în ceea ce privește performanța detectorului, schimbând atât dimensiunea imaginii de intrare, cât și varietatea ancorelor(Subsecțiunea 5.1.2) folosite în detecție. În Tabelul 6.2 se poate observa cum o dimensiunea mai mică a imaginii de intrare și mai puține ancore folosite fac ca mAP-ul să fie cu aproximativ 5% mai mic. De asemenea, după cum am precizat în Secțiunea 2.3, am experimentat performanța detectorului cu ResNet-ul folosit doar ca extractor, fără a mai fi antrenat și acesta, iar performanța a scăzut cu 0.56%, conform Tabelei 6.1.

Model	mAP
Detector ResNet18-FE	96.58%
Detector ResNet18	<b>97.14%</b>

Tabela 6.1: *mAP*-ul pentru detectorul folosind ResNet18 cu straturile înghețate vs folosind ResNet18 care își schimbă ponderile odată cu antrenarea acestuia.

Model	Dimensiune imagine de intrare	Dimensiuni ancore	Raporturi de aspect ancore	mAP
Detector ResNet18	900	16, 32, 64, 128, 256, 512 (px)	0.25, 0.5, 1.0, 1.5, 2.0, 2.5	<b>97.14%</b>
Detector ResNet18	700	32, 64, 128, 256, 512 (px)	0.5, 1.0, 2.0	92.19%

Tabela 6.2: Variația *mAP*-ului în funcție de dimensiunea imaginii de intrare și aspectul ancorelor.

În Capitolul 5 am prezentat cele trei abordări propuse, enunțând faptul că cea de-a doua abordare mi-a adus cele mai bune rezultate(detector mașini + clasificator modele). În Tabelul 6.3 se poate observa cum prima metodă(detector modele de mașini) este cu mult inferioară din punctul de vedere al performanței.

Model	mAP
Detector ResNet18 modele mașini	81.84%
Detector mașini ResNet18 + Clasificator modele	<b>97.14%</b>

Tabela 6.3: *mAP* pentru Metoda 1 vs Metoda 2, descrise în Capitolul 5

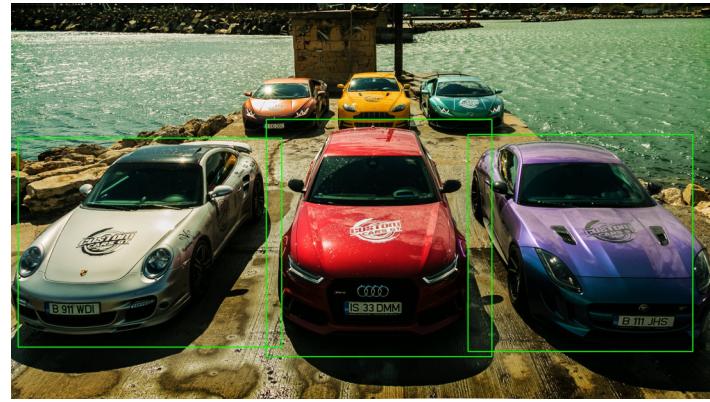


Figura 6.4: Problemele detectorului cu mașinile în dimensiuni mici.

### 6.2.1 Grafice antrenament

În antrenarea detectorului de mașini, scopul urmărit a fost minimizarea loss -ului până sub un prag cu valoarea de aproximativ 0.05.

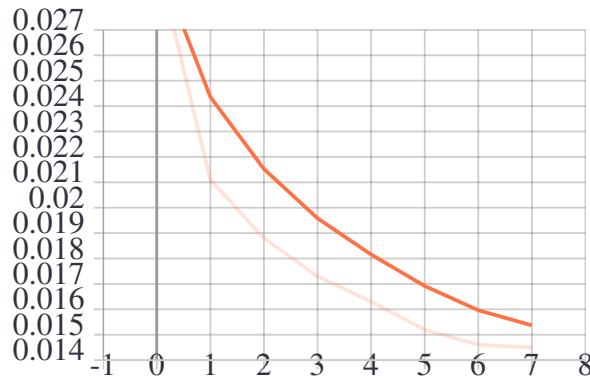


Figura 6.5: Loss-ul regresiei din RPN pe datele de antrenament



Figura 6.6: Loss-ul clasificatorului din RPN pe datele de antrenament

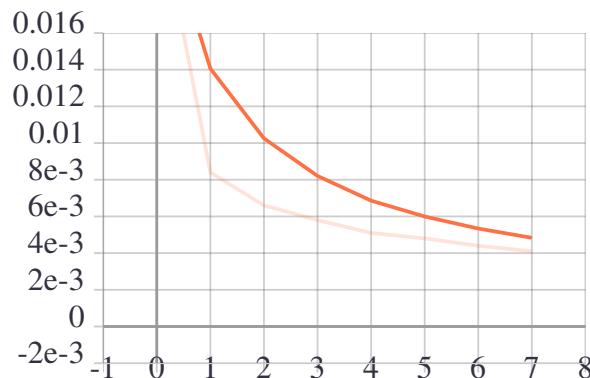


Figura 6.7: Loss-ul regresiei pe bounding box-uri pe datele de antrenament

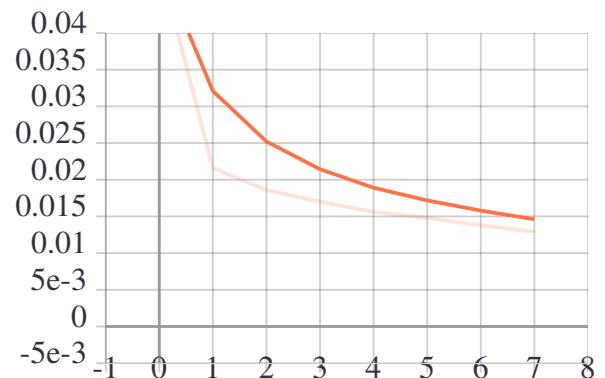


Figura 6.8: Loss-ul clasificatorului pe datele de antrenament

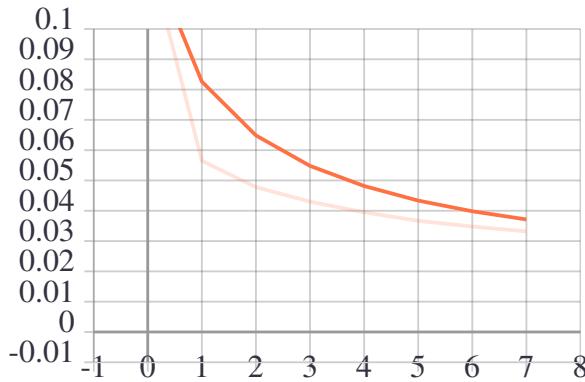


Figura 6.9: Loss-ul total pe datele de antrenament

### 6.3 Clasificarea modelelor de mașini

După ce mașinile sunt detectate, acestea trebuie clasificate. În elaborarea prezentei lucrări, au avut loc multiple experimente pentru optimizarea clasificării. În primul rând, am vrut să văd diferențele de performanță dintre ResNet34 și ResNet50. După cum se poate observa în Tabelul 6.5, ResNet50, antrenat pentru doar 25 de epoci față de 50, a avut o eroare mai mică cu 0.52 față de ResNet34, dacă ne luăm după prima predicție a clasificării, sau cu 0.2 mai mică dacă ne luăm după primele cinci predicții. Însă, diferența în numărul de parametrii dintre cele două arhitecturi este mare(cu aproximativ 4 milioane mai mulți parametrii pentru ResNet50), fapt ce m-a dus la decizia de a face un trade-off între timpul de rulare și acuratețe. Astfel, ResNet34 a devenit arhitectura de bază pentru sarcina de clasificare. În al doilea rând, am încercat să folosesc această rețea ca extractor de caracteristici(detalii în Secțiunea 2.3), însă, conform Tabelei 6.4 variantele în care toate ponderile sunt modificate au dat rezultate mult mai bune(între ResNet34 și ResNet34-FE este o diferență de 0.72 între erorile pentru prima predicție, în favoarea primei arhitecturi). Acest lucru se întâmplă deoarece, chiar dacă în ImageNet rețeaua a "văzut" și imagini cu mașini, sarcina curentă este una mult mai specifică, modelele de mașini fiind mai greu de clasificat.

De asemenea, după cum se poate observa tot în Tabela 6.4, arhitecturile în care am adăugat mecanismele de atenție(detaliile în Subsecțiunea 5.2.4) nu au reușit să depășească performanța arhitecturii fără aceste mecanisme.

Problemele care apar destul de des în clasificarea modelelor de mașini sunt cauzate de faptul că unele clase sunt foarte asemănătoare cu alte clase în anumite ipostaze(de exemplu,

mașinile *Convertible* se pot confunda cu cele *Coupe*, atunci când primele nu sunt decapotate). Astfel, în Figurile 6.10, 6.11 și 6.12 se poate observa cum toate cele trei variații ale arhitecturilor destinate clasificării, dau o primă predicție greșită, însă clasa corectă se află în top 5. De asemenea, prin *heatmap*-ul (detalii în Subsecțiunea 5.3.1) corespunzător primei clase prezise se poate observa și ce l-a făcut pe acesta să dea un astfel de rezultat. În Figura 6.13 sunt expuse acele clase cu care clasa corectă este confundată de către diversele arhitecturi.



Figura 6.10: Exemplu din clasa Aston Martin Virage Convertible 2012 clasificat greșit ca fiind Aston Martin V8 Vantage Coupe 2012 de către ResNet34.



Figura 6.11: Exemplu din clasa Aston Martin Virage Convertible 2012 clasificat greșit ca fiind Aston Martin V8 Vantage Coupe 2012 de către SEResNet34



### Top five predictions

1. ASTON MARTIN V8 VANTAGE COUPE 2012 74.6379%
2. ASTON MARTIN VIRAGE COUPE 2012 4.7561%
3. BENTLEY CONTINENTAL GT COUPE 2007 3.8811%
4. MERCEDES-BENZ SL-CLASS COUPE 2009 3.6857%
5. ASTON MARTIN VIRAGE CONVERTIBLE 2012 2.3377%

Figura 6.12: Exemplu din clasa Aston Martin Virage Convertible 2012 clasificat greșit ca fiind Aston Martin V8 Vantage Coupe 2012 de către CBAMResNet34

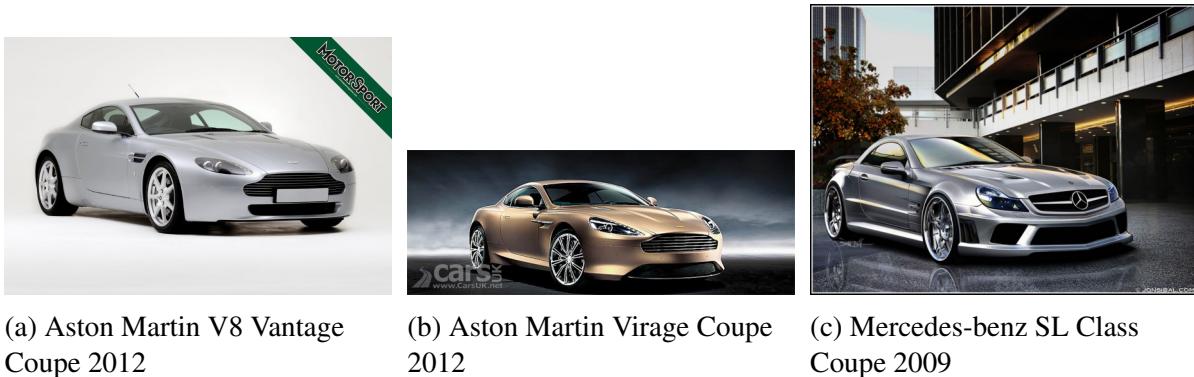


Figura 6.13: Modele cu care modelul din Figurile 6.10, 6.11 și 6.12 poate fi confundat ușor, chiar și de o persoană fără prea multă experiență.

Model	Număr total de parametrii	Număr de parametrii antrenabili	Eroare-Top1	Eroare-Top5	Număr epoci antrenament
ResNet34	21.365.213	21,365,213	<b>6.6320</b>	0.8389	50
SEResNet34	21,526,409	21,526,409	6.8318	0.7990	
CBAMResNet34	21,689,173	21,689,173	7.7506	0.9188	
ResNet34-FE	21,365,213	20,020,253	7.3511	<b>0.7590</b>	
SEResNet34-FE	21,526,409	20,181,449	8.1102	0.9988	
CBAMResNet34-FE	21,689,173	20,344,213	8.3499	0.9988	

Tabela 6.4: Rata de eroare(%) pentru subsetul de testare al setului de date. "FE" înseamnă *feature-extractor*(detalii în Secțiunea ??).

Model	Număr total de parametrii	Număr de parametrii antrenabili	Eroare-Top1	Eroare-Top5	Număr epoci antrenament
ResNet34	21.365.213	21,365,213	6.6320	0.8389	50
ResNet50	25,557,032	25,557,032	<b>6.1126</b>	<b>0.6392</b>	25

Tabela 6.5: Rata de eroare(%) pentru subsetul de testare al setului de date.

Model	Acuratetea medie	aston_martin_v8_vantage_convertible_2012	audi_s5_coupe_2012	bmw_x3_suv_2012	hyundai_sonata_sedan_2012	tesla_model_s_sedan_2012	volkswagen_golf_hatchback_2012	volvo_xc90_suv_2007
ResNet34	<b>93.37</b>	76.19	58.82	<b>100</b>	<b>93.33</b>	<b>100</b>	<b>100</b>	<b>100</b>
SEResNet34	93.17	72.73	66.67	<b>100</b>	<b>93.33</b>	<b>100</b>	<b>100</b>	94.44
CBAMResNet34	92.25	<b>80.95</b>	64.29	93.33	<b>93.33</b>	<b>100</b>	<b>100</b>	<b>100</b>
ResNet34-FE	92.65	71.43	66.67	<b>100</b>	92.86	<b>100</b>	<b>100</b>	<b>100</b>
SEResNet34-FE	91.89	80	<b>71.43</b>	<b>100</b>	92.86	<b>100</b>	<b>100</b>	<b>100</b>
CBAMResNet34-FE	91.65	75	60	<b>100</b>	<b>93.33</b>	93.75	94.44	94.12

Tabela 6.6: Acuratețea(%)medie pentru subsetul de testare al setului de date și acuratețea pe unele clase.

### 6.3.1 Grafice antrenament/validare

Validarea antrenării clasificatorului a fost făcută pe 10% din setul de antrenament, micșorând rata de învățare cu un factor de 10 atunci când *loss*-ul pe validare nu scade în mai mult de 5 epoci.

#### ResNet34

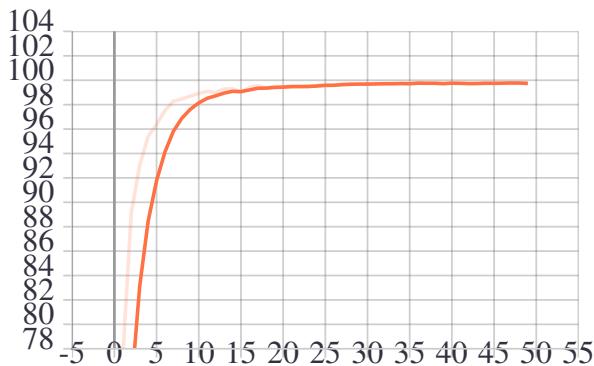


Figura 6.14: Acuratețea pe datele de antrenament ResNet34

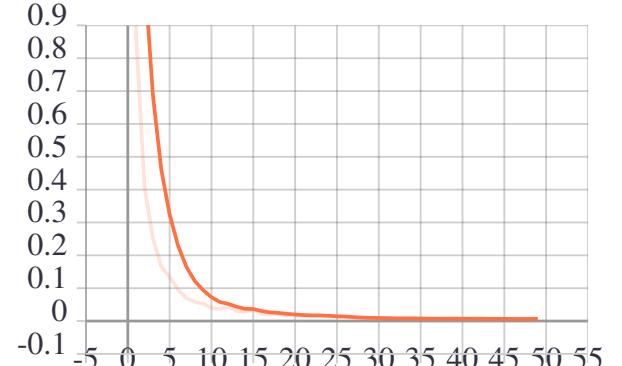


Figura 6.15: Loss-ul pe datele de antrenament ResNet34

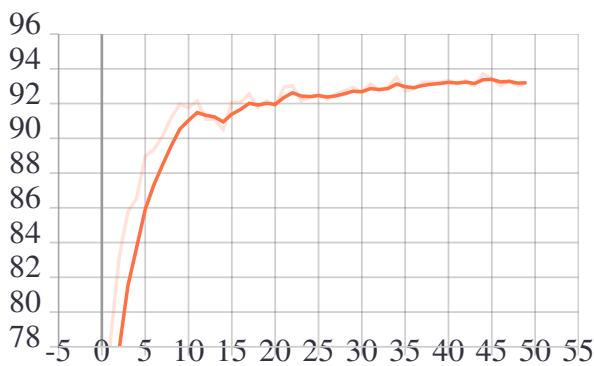


Figura 6.16: Acuratețea pe datele de validare ResNet34

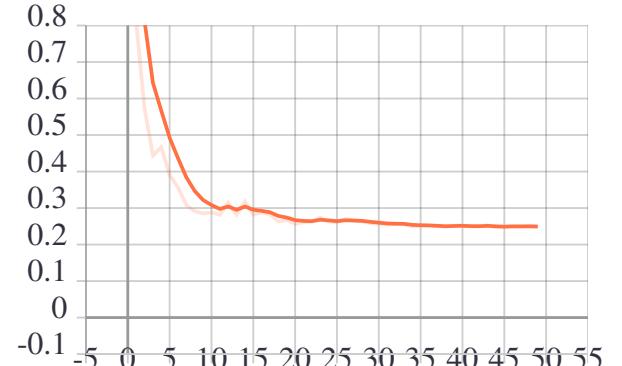


Figura 6.17: Loss-ul pe datele de validare ResNet34

# Capitolul 7

## Descrierea aplicației

### 7.1 Interfața web

Interfața web disponibilă pentru această lucrare de licență este una destul de simplistă, de tip *one page*, al cărei template este inspirat din [9]. Aceasta este bazată în mare parte pe elemente de Bootstrap, a căror detalii se pot găsi în Capitolul 4.

În momentul în care utilizatorul intră prima oară pe pagină, acesta are fie opțiunea de a încărca o imagine, fie de a schimba arhitectura cu care vrea să facă clasificarea imaginilor(Figura 7.4).

Există două posibilități de a încărca o imagine: alegerea acesteia din memoria locală sau alegerea uneia disponibile pe server(Figura 7.1). În momentul alegerii unei imagini, aceasta este trimisă automat server-ului pentru a face predicția pe ea. Prima etapă este detectarea tuturor mașinilor din imagine. Apoi, aceste detectii sunt trecute prin clasificatorul pentru modelele mașinilor. Aceste două etape sunt detaliate în Capitolul 5. În timpul detectiei, utilizatorul are feedback în timp real de la server, deoarece pe ecran îi apare un element *modal* care îl anunță că inferența încă are loc. Când aceasta este gata, *modal*-ul se închide și pe ecran îi apar rezultatele. Mașinile sunt marcate cu un bounding box de culoare verde în prima imagine din *carousel*-ul cu rezultate. Apoi, în același *carousel*, sunt disponibile, două câte două, partea din imaginea originală cu mașina detectată și *heatmap*-ul corespunzător, împreună cu modelul mașinii. În partea dreaptă este disponibil top-ul predicțiilor pentru acea mașină, ordonat descrescător în funcție de acuratețe.

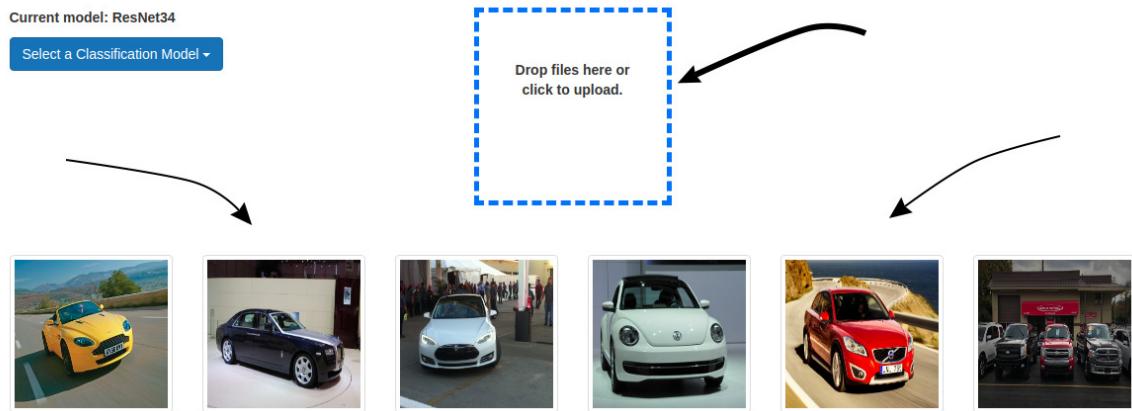


Figura 7.1: Interfața web în momentul primei intrări.

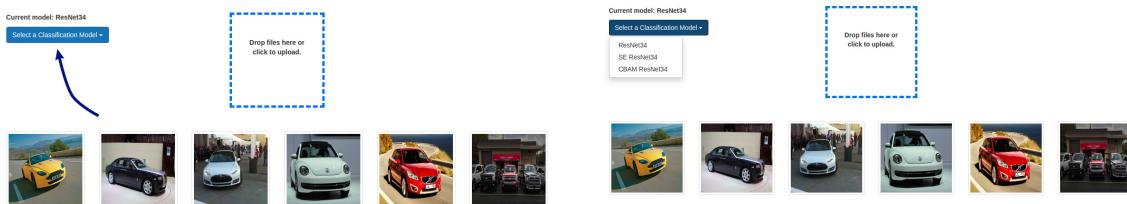


Figura 7.2: Posibilitatea de a schimba modelul folosit în clasificare.

Figura 7.3: Modelele disponibile sunt: ResNet34, SEResNet34 si CBAMResNet34.

Figura 7.4: Schimbarea modelului folosit pentru clasificare.

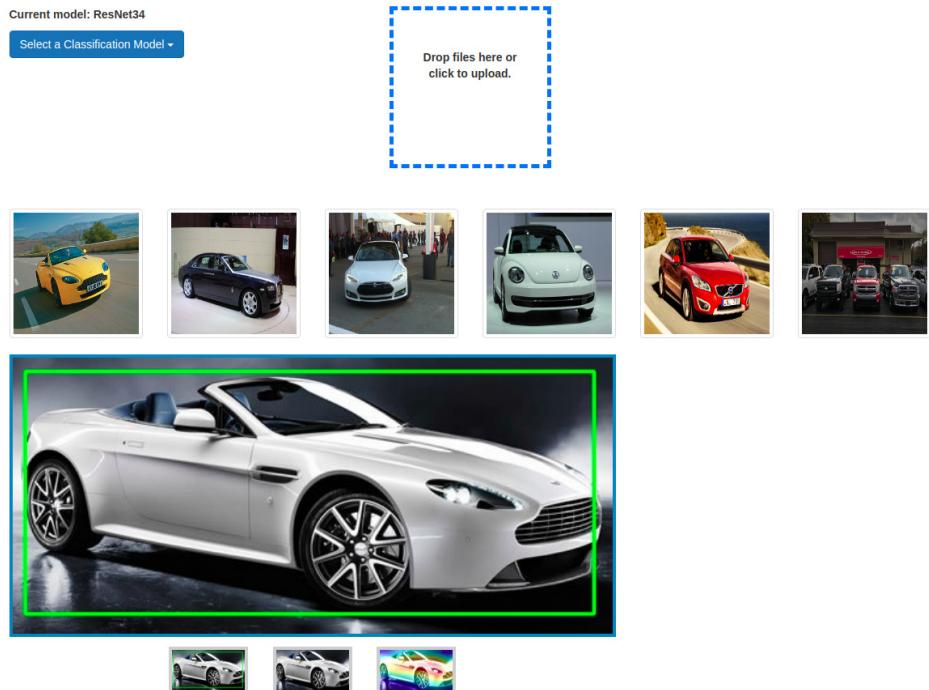


Figura 7.5: Detectiile de mașini.

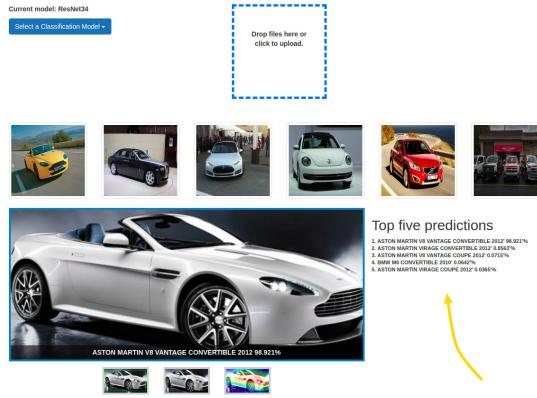


Figura 7.6: Clasificarea unei mașini în funcție de model.

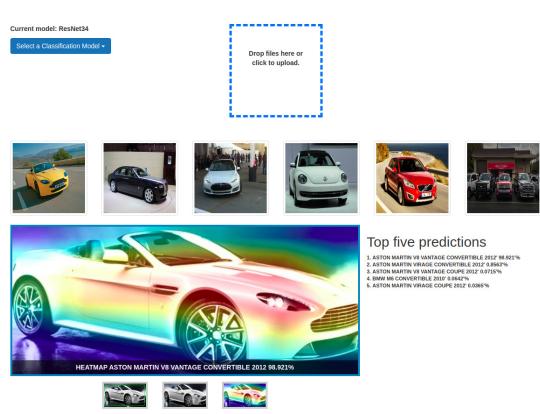


Figura 7.7: Heatmap-ul unei clasificări.

Figura 7.8: În partea stangă a paginii: rezultatele clasificării. În partea dreaptă: top-ul predicțiilor ordonate descrescător.

## 7.2 Aplicația de bază

Aplicația de bază poate fi integrată foarte ușor pe un server, ca în prezența lucrare de licență, sau pe orice dispozitiv care ar avea nevoie de aceste detecții cu modelele mașinilor(de exemplu camerele video de supraveghere din trafic). Server-ul preprocesează imaginile primite, redimensionându-le la 900x900 de pixeli și apoi normalizându-le cu media și deviația standard cu care arhitectura a fost antrenată, aceleași care sunt folosite și în [5]. După ce detectorul de mașini este aplicat pe imagine, detecțiile sunt trecute prin algoritmul *NMS*, pentru a fi sigur că o mașină nu a fost detectată de mai multe ori. Apoi, detecțiile rămase sunt redimensionate la 224x244 de pixeli și trecute, pe rând, prin clasificator. În acest moment, imaginea principală, pe care sunt desenate cu verde bounding box-urile cu mașini, imaginile tăiate, în care se află câte o mașină, *heatmap*-ul corespunzător și predicțiile pentru fiecare mașină, sunt trimise *front-end*-ului. Aici, toate aceste informații sunt organizate după cum am descris în secțiunea anterioară.

# Capitolul 8

## Concluzii și dezvoltări ulterioare

Prezenta lucrare de licență a propus un sistem pentru detecția mașinilor din imagini și clasificarea acestora în funcție de model. Sistemul dezvoltat este bazat în mare parte pe aplicația de bază, cea care se ocupă cu preprocesarea imaginilor la primire, cu predicțiile făcute și cu calcularea *heatmap*-urilor, cu scopul de a înțelege mai bine ce a determinat sistemul să ia o astfel de decizie pentru o anumită clasă. Peste această aplicație de bază există interfața web ce are un rol demonstrativ, astfel utilizatorul putând observa cum funcționează sistemul.

Aplicația de bază poate fi oricând integrată cu un alt sistem deja disponibil(un site pentru vânzarea mașinilor, camerele video de supraveghere).

Sistemul nu este perfect, dând rateuri uneori când este pus să clasifice o imagine în care se află o mașină dintr-o anumită clasă, însă în clasele disponibile există și altele cu caracteristici foarte asemănătoare(vezi Secțiunea 6.3). Acest lucru poate apărea și în cazul oamenilor foarte des, întâmplându-se de multe ori ca o persoană să nu poată distinge între două modele de mașini. În ceea ce privește detecția de mașini, detectorul ar putea fi îmbunătățit prin antrenarea lui pe mai multe imagini de antrenare, în care mașinile să apară în diferite dimensiuni. Această îmbunătățire ar ajuta contextul în care mașinile vizate nu sunt în prim plan, ci într-un plan secund și astfel nu sunt de dimensiuni foarte mari.

O altă optimizare ce ar putea fi adusă întregului sistem ar fi înlocuirea detectorului în două stagii(Faster R-CNN, detalii în Secțiunea 5.1) cu un detector ce rulează în doar un stagiu, cum ar fi SSD[20] sau YOLO[29]. Această modificare ar reduce timpul total de inferență și ar fi mai potrivit pentru detecția în timp real.

# Bibliografie

- [1] *AI Blog*. Accesat 14 iunie 2020. URL: <http://www.pabloruizruiz10.com/aiblog/aiblog.html>.
- [2] G. Bradski. “The OpenCV Library”. in: *Dr. Dobb’s Journal of Software Tools* (2000).
- [3] *CS231n: Convolutional Neural Networks for Visual Recognition*. Accesat 10 mai 2020. URL: <http://cs231n.stanford.edu/>.
- [4] Deepbear. *Pytorch car classifier - 90% accuracy*. Accesat 08 aprilie 2020. **april 2019**. URL: <https://www.kaggle.com/deepbear/pytorch-car-classifier-90-accuracy>.
- [5] J. Deng **and others**. “ImageNet: A Large-Scale Hierarchical Image Database”. in: *CVPR09*. 2009.
- [6] Mohit Deshpande. *Perceptrons: The First Neural Networks*. Accesat 20 mai 2020. **may 2020**. URL: <https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/>.
- [7] *Dropzone*. Accesat 11 martie 2020. URL: <https://flask-dropzone.readthedocs.io/en/latest/>.
- [8] Mark Everingham **and others**. “The Pascal Visual Object Classes (VOC) Challenge”. in: *Int. J. Comput. Vision* 88.2 (**june** 2010), **pages** 303–338. ISSN: 0920-5691. DOI: [10.1007/s11263-009-0275-4](https://doi.org/10.1007/s11263-009-0275-4). URL: <https://doi.org/10.1007/s11263-009-0275-4>.
- [9] *Face Recognition, Vehicle Recognition, Face Detection Person Detection API*. Accesat 04 martie 2020. URL: <https://www.sighthound.com/products/cloud>.

- [10] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO - Object Detection Algorithms*. Accesat 20 mai 2020. **july** 2018. URL: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e>.
- [11] Ross B. Girshick. “Fast R-CNN”. in: *CoRR* abs/1504.08083 (2015). arXiv: [1504 . 08083](https://arxiv.org/abs/1504.08083). URL: <http://arxiv.org/abs/1504.08083>.
- [12] Ross B. Girshick **and others**. “Rich feature hierarchies for accurate object detection and semantic segmentation”. in: *CoRR* abs/1311.2524 (2013). arXiv: [1311 . 2524](https://arxiv.org/abs/1311.2524). URL: <http://arxiv.org/abs/1311.2524>.
- [13] Kaiming He **and others**. “Deep Residual Learning for Image Recognition”. in: *CoRR* abs/1512.03385 (2015). arXiv: [1512 . 03385](https://arxiv.org/abs/1512.03385). URL: <http://arxiv.org/abs/1512.03385>.
- [14] Kaiming He **and others**. “Mask R-CNN”. in: *CoRR* abs/1703.06870 (2017). arXiv: [1703 . 06870](https://arxiv.org/abs/1703.06870). URL: <http://arxiv.org/abs/1703.06870>.
- [15] Jie Hu, Li Shen **and** Gang Sun. “Squeeze-and-Excitation Networks”. in: *CoRR* abs/1709.01507 (2017). arXiv: [1709 . 01507](https://arxiv.org/abs/1709.01507). URL: <http://arxiv.org/abs/1709.01507>.
- [16] Sergey Ioffe **and** Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. in: *CoRR* abs/1502.03167 (2015). arXiv: [1502 . 03167](https://arxiv.org/abs/1502.03167). URL: <http://arxiv.org/abs/1502.03167>.
- [17] Kamwoh. *kamwoh/Car-Model-Classification*. Accesat 07 aprilie 2020. URL: [https : //github . com /kamwoh /Car - Model - Classification](https://github.com/kamwoh/Car-Model-Classification).
- [18] Jonathan Krause **and others**. “3D Object Representations for Fine-Grained Categorization”. in: *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*. Sydney, Australia, 2013.
- [19] Alex Krizhevsky, Ilya Sutskever **and** Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. in: *Advances in Neural Information Processing Systems 25*. **byeditor**F. Pereira **and others**. Curran Associates, Inc., 2012, **pages** 1097–1105. URL: [http : / / papers . nips . cc / paper / 4824 - imagenet - classification - with - deep - convolutional - neural - networks . pdf](http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf).

- [20] Wei Liu **and others**. “SSD: Single Shot MultiBox Detector”. in: *CoRR* abs/1512.02325 (2015). arXiv: 1512.02325. URL: <http://arxiv.org/abs/1512.02325>.
- [21] Wes McKinney. “Data Structures for Statistical Computing in Python”. in: *Proceedings of the 9th Python in Science Conference*. byeditor Stéfan van der Walt **and** Jarrod Millman. 2010, pages 56–61. DOI: 10.25080/Majora-92bf1922-00a.
- [22] Mark Otto **and** Jacob Thornton. *Introduction*. Accesat 13 martie 2020. URL: <https://getbootstrap.com/docs/4.5/getting-started/introduction/>.
- [23] Adam Paszke **and others**. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. in: *Advances in Neural Information Processing Systems 32*. byeditor H. Wallach **and others**. Curran Associates, Inc., 2019, pages 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [24] Fabian Pedregosa **and others**. “Scikit-learn: Machine Learning in Python”. in: *Journal of Machine Learning Research* 12.85 (2011), pages 2825–2830. URL: <http://jmlr.org/papers/v12/pedregosall1a.html>.
- [25] Ian Pointer. *Programming PyTorch for deep learning: creating and deploying deep learning applications*. OReilly, 2019.
- [26] Ning Qian. “On the momentum term in gradient descent learning algorithms”. in: *Neural Networks* 12.1 (1999), pages 145–151. ISSN: 0893-6080. DOI: [https://doi.org/10.1016/S0893-6080\(98\)00116-6](https://doi.org/10.1016/S0893-6080(98)00116-6). URL: <http://www.sciencedirect.com/science/article/pii/S0893608098001166>.
- [27] Sergio Lima Netto Rafael Padilla **and** Eduardo A. B. da Silva. “Survey on Performance Metrics for Object-Detection Algorithms”. in: (2020).
- [28] Haripriya Reddy. *Insight into Faster R-CNN for Object Detection*. Accesat 14 aprilie 2020. january 2020. URL: <https://medium.com/analytics-vidhya/insight-into-faster-r-cnn-for-object-detection-f1e64240eee1>.
- [29] Joseph Redmon **and others**. “You Only Look Once: Unified, Real-Time Object Detection”. in: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: <http://arxiv.org/abs/1506.02640>.

- [30] Shaoqing Ren **and others**. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. in: *CoRR* abs/1506.01497 (2015). arXiv: [1506 . 01497](https://arxiv.org/abs/1506.01497). URL: <http://arxiv.org/abs/1506.01497>.
- [31] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. in: *CoRR* abs/1609.04747 (2016). arXiv: [1609 . 04747](https://arxiv.org/abs/1609.04747). URL: <http://arxiv.org/abs/1609.04747>.
- [32] Shibani Santurkar **and others**. “How Does Batch Normalization Help Optimization?” in: *Advances in Neural Information Processing Systems 31*. **byeditorS.** Ben-gio **and others**. Curran Associates, Inc., 2018, **pages** 2483–2493. URL: <http://papers.nips.cc/paper/7515-how-does-batch-normalization-help-optimization.pdf>.
- [33] Ramprasaath R. Selvaraju **and others**. “Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization”. in: *CoRR* abs/1610.02391 (2016). arXiv: [1610 . 02391](https://arxiv.org/abs/1610.02391). URL: <http://arxiv.org/abs/1610.02391>.
- [34] Karen Simonyan **and** Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. in: *CoRR* abs/1409.1556 (2014). URL: <http://arxiv.org/abs/1409.1556>.
- [35] E. Stevens, L. Antiga **and** T. Viehmann. *Deep Learning with PyTorch*. Manning Publications, 2020. ISBN: 9781617295263. URL: [https : //books . google . ro / books ? id = 89BlwwEACAAJ](https://books.google.ro/books?id=89BlwwEACAAJ).
- [36] J.R.R. Uijlings **and others**. “Selective Search for Object Recognition”. in: *International Journal of Computer Vision* (2013). DOI: [10 . 1007 / s11263 - 013 - 0620 - 5](https://doi.org/10.1007/s11263-013-0620-5). URL: <http://www.huppelen.nl/publications/selectiveSearchDraft.pdf>.
- [37] S. van der Walt, S. C. Colbert **and** G. Varoquaux. “The NumPy Array: A Structure for Efficient Numerical Computation”. in: *Computing in Science Engineering* 13.2 (2011), **pages** 22–30.
- [38] Guido Van Rossum **and** Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

- [39] *Welcome to Flask*. Accesat 03 martie 2020. URL: <https://flask.palletsprojects.com/en/1.1.x/>.
- [40] Wikipedia. *Bilinear interpolation* — Wikipedia, The Free Encyclopedia. <http://en.wikipedia.org/w/index.php?title=Bilinear%20interpolation&oldid=957374633>. Accesat 14 iunie 2020. 2020.
- [41] Wikipedia. *Gradient descent* — Wikipedia, The Free Encyclopedia. <http://en.wikipedia.org/w/index.php?title=Gradient%20descent&oldid=962649577>. Accesat 17 mai 2020. 2020.
- [42] Sanghyun Woo **and others**. “CBAM: Convolutional Block Attention Module”. in: *CoRR* abs/1807.06521 (2018). arXiv: <1807.06521>. URL: <http://arxiv.org/abs/1807.06521>.
- [43] Bolei Zhou **and others**. “Learning Deep Features for Discriminative Localization”. in: *CoRR* abs/1512.04150 (2015). arXiv: <1512.04150>. URL: <http://arxiv.org/abs/1512.04150>.
- [44] Zhengxia Zou **and others**. “Object Detection in 20 Years: A Survey”. in: *CoRR* abs/1905.05055 (2019). arXiv: <1905.05055>. URL: <http://arxiv.org/abs/1905.05055>.