



UNIVERSITATEA DIN BUCUREȘTI



FACULTATEA
DE
MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

LUCRARE DE DISERTAȚIE
EXECUȚIA WORKFLOW-URILOR DURABILE ÎN
CLOUD

Absolvent

Moldovan George-Alexandru

Coordonator științific

Prof. dr. Letiția Marin

București, septembrie 2022

Rezumat

Considerând trend-ul ascendent al arhitecturii orientate spre microservicii și migrarea de la vechiul mod de dezvoltare monolit, această lucrare de disertație își propune analiza soluțiilor prezente în piața în ceea ce privește managementul unei arhitecturi pe microservicii în cloud, și analiza unor contribuții personale aduse unui framework open-source de gestionare a workflow-urilor în cloud. Sistemele informatice preiau din complexitatea operațiilor de zi cu zi, astfel că arborii de decizie trebuie interpretați și gestionați în mod corect pentru a realiza un sistem care să îndeplinească cerințele de piață curente. Pentru o lungă perioadă de timp, problema gestionării tranzacțiilor distribuite și management-ul workflow-urilor a fost inexistentă, deoarece într-un sistem monolit, caracteristicile tranzacționale ale bazelor de date ce susțineau astfel de sisteme erau îndeajuns pentru a avea toate garanțiile necesare astfel încât sistemul să fie mereu lăsat într-o stare consistentă. Microserviciile și arhitecturile distribuite în general, deși vin cu o serie lungă de avantaje, poate cel mai greu de gestionat lucru este management-ul stării unei acțiuni, atunci când aceasta se întinde pe mai multe microservicii, și pe o durată lungă de timp.

Abstract

Considering the upward trend of microservices-oriented architecture and the migration from the old monolithic development mode, this dissertation aims to analyze the solutions present in the market in terms of managing a microservices architecture in the cloud, and the analysis of personal contributions to an open-source framework for managing cloud workflows. Information systems take over the complexity of day-to-day operations, so decision trees must be interpreted and managed correctly to achieve a system that meets current market requirements. For a long time, the problem of distributed transaction management and workflow management was non-existent, because in a monolithic system, the transactional characteristics of the databases that supported such systems were sufficient to have all the necessary guarantees so that the system should always be left in a consistent state. Distributed microservices and architectures in general, although they come with a long list of advantages, perhaps the most difficult thing to manage is the management of the state of an action, when it extends over several microservices, and over a long period of time.

Cuprins

1	Introducere	3
1.1	Motivatie	3
1.2	Context	4
1.3	Alte abordări	5
1.4	Conținutul lucrării	6
2	Analiza arhitecturii si a tehnologiilor folosite	7
2.1	Etapa de antrenare	7
2.1.1	Autoencoder	7
2.1.2	Clusterizare K-means	8
2.1.3	Clasificare multi-class folosind SVM	9
2.2	Etapa de inferență	10
2.3	Etapa de lansare în cloud	10
2.4	Tehnologii folosite în dezvoltarea interfeței	13
3	Sistemul de detecție a anomaliilor	15
3.1	Detecția obiectelor	15
3.2	Analiza antrenării sistemului	16
3.3	Analiza etapei de inferență	23
3.4	Scenarii de utilizare	26
4	Execuția în cloud a sistemului	28
4.1	Cloud-computing în inteligență artificială	28
4.2	Analiza opțiunilor	29
4.3	Comparație între soluțiile PaaS si FaaS	31
4.4	Descrierea soluției curente	32
5	Rezultate și Concluzii	34
5.1	Timpi de execuție și necesarul de resurse	35
5.2	Concluzii	36

Capitolul 1

Introducere

1.1 Motivatie

Detectarea anomaliilor în video este în strânsă legătură cu sistemele de supraveghere inteligente, un domeniu care a fost și este de interes pentru mine. La rândul lor, sistemele de supraveghere inteligente, au o mare importanță în securitatea publică. Cu toții ne dorim o lume în care apelurile de urgență în caz de incendiu se fac automat, alunecările de teren sunt descoperite înainte să fie prea târziu, iar oamenii rău intenționați sunt opriți înainte să se întâmple tragedii.

Astfel, arhitectura folosită se bazează pe detecția caracteristicilor spațio-temporale ale evenimentelor prezente în video, care mai apoi sunt împărțite în clase de normalitate. Aceste caracteristici sunt extrase trecând evenimentul printr-o serie de autoencodere pentru a folosi mai apoi reprezentarea latentă în cadrul clasificării finale. În etapa de antrenare, se folosesc filmări ce prezintă comportamentul normal în scenariul analizat. Analizând aceste video-uri putem crea un model capabil să recunoască dacă un eveniment aparține unei clase de normalitate analizate până acum, sau dacă este un caz anormal. Deoarece un eveniment poate fi ales în multe moduri, în cadrul acestui sistem un eveniment reprezintă orice obiect aflat în cadru. Analiza asupra fiecarui obiect conține și un cadru precedent dar și unul viitor, asemănător cu modalitatea folosită de *Ionescu et al.* [?]. O mențiune în acest sens ar fi că pentru fiecare obiect sunt analizate și cadrele de la pozițiile $t-3$ și $t+3$ respectiv la poziția t a cadrului analizat. Din acest motiv, atunci când sistemul va analiza un video în sistem live-feed, analiza se va face cu un decalaj de 3 cadre. Având în vedere că pentru un video actual viteza de redare este de minim 15 cadre pe secundă, acest decalaj este neglijabil.

Pe lângă partea algoritmică a detectării anomaliilor, o altă arie de interes a acestei lucrări este cloud computing. Această parte analizează un nou mod de rulare, ce facilitează atât dezvoltarea cât și execuția ulterioară a unor sisteme complexe. Acest nou mod constă în folosirea unei arhitecturi plasate în cloud, ce oferă dezvoltatorului posibilitatea să creeze sisteme ce necesită multe resurse în timpul rulării, fără costurile asociate creerii și menținerii unei infrastructuri proprii. Pe de altă parte, având în

vedere că toate operațiunile sunt executate în cloud, utilizatorii serviciului au nevoie doar de conexiune la internet și cerințe minime pentru sistemele proprii, fără a fi nevoiți să achiziționeze echipamente noi pentru a folosi sisteme de detecție a anomaliilor.

1.2 Context

Detectarea anomaliilor în video poate fi văzută ca o problemă subiectivă, deoarece un eveniment este normal sau anormal doar dacă este luat în considerare și contextul în care acesta apare. Un exemplu foarte bun este comparația între două persoane care se luptă și o persoană care se plimbă. Care dintre aceste evenimente este anormal? Desigur, depinde de context. Dacă sistemul supraveghează o arenă de lupte, atunci persoana care se plimbă în ring prezintă un comportament anormal, în timp ce luptătorii prezintă comportamentul așteptat. Din acest motiv majoritatea lucrărilor din domeniu [?, ?, ?]... abordează un mod de lucru bazat pe antrenarea folosind video-uri ce provin din aceeași locație cu cele de test. Tocmai din cauza dependenței de context, detectarea anomaliilor nu este o problemă ce poate fi generalizată, astfel fiecare scenariu necesită o antrenare și un model propriu.

Ca și moduri de expunere a soluției software către utilizatori, aceasta se poate face în 2 moduri :

- Folosind servere proprii
- Folosind servicii cloud

Folosirea serverelor proprii presupune, pe lângă prezența fizică a serverelor și cumpărarea tuturor serviciilor conexe, cum ar fi servere de baze de date, sisteme de distribuire a fișierelor, infrastructură de rețea, ș.a.m.d., este nevoie și de o echipă dedicată pentru întreținerea infrastructurii și repararea eventualelor probleme ce pot apărea. Aceste considerente, împreună cu faptul că scalarea soluțiilor software este foarte anevoioasă atunci când este folosită infrastructura proprie, fac această soluție să nu mai fie folosită în mod curent deoarece încetinește dezvoltarea aplicației, iar rezultatul final este și el unul mai puțin calitativ decât ce se poate obține folosind soluții în cloud.

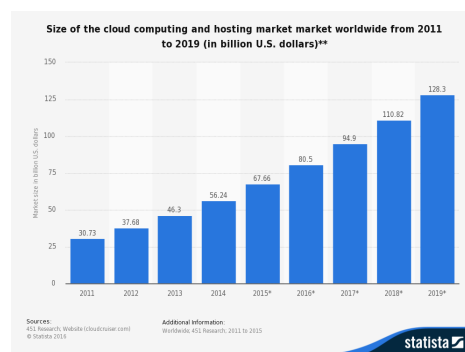


Figura 1.1: Statistică ce evidențiază importanța domeniului cloud computing în ultimii ani

Sursă: <https://www.statista.com/>

Folosirea serviciilor cloud oferă multiple posibilități de dezvoltare a aplicațiilor, de la crearea unei infrastructuri complete care este administrată în totalitate de către dezvoltator, la medii de execuție serverless care sunt complet administrate de către provider-ul de servicii cloud. Deși crearea unei infrastructuri proprii încă este necesară acolo unde legislația nu permite ca datele să fie stocate în cloud, în toate celelalte cazuri se face migrarea spre soluții cloud, lucru vizibil și în evoluția cotei de piață a domeniului cloud-computing la nivel global, așa cum se poate observa și în figura 1.1.

1.3 Alte abordări

Detectarea anomaliilor vizuale este un subiect intens cercetat, numeroase lucrări fiind publicate având aceasta arie de interes. În general, detectarea anomaliilor este abordată din perspectiva învățării evenimentelor normale ca mai apoi să fie detectate evenimentele ce nu se potrivesc șabloanelor create folosind evenimentele normale [?, ?, ?, ?].

Nivelul de supervizare variază destul de mult în abordările existente, fiind atât lucrări ce definesc normalitatea pe baza unor constrângeri [?, ?], dar și metode ce se bazează complet pe învățarea nesupervizată a comportamentului normal [?, ?].

Antic et al. [?] folosește o arhitectură în două etape, ce propune evaluarea unui video prin folosirea unui algoritm de separarea a foreground ului de background, pe baza faptului că filmarea este făcută dintr-un punct static. Prin această metoda, acesta obține o serie de *ipoteze* ce urmează mai apoi vor fi analizate și clasificate prin metode statistice folosind caracteristici spațio-temporale, dar și legăturile dintre acestea.

Lu et al. [?] folosește o reprezentare a evenimentelor folosind o structură piramidală ce reprezintă caracteristicile spațio-temporale ale evenimentelor prezente într-un cadru. Această reprezentare este obținută prin scalarea imaginii la diferite dimensiuni și împărțirea acesteia în regiuni ce nu se suprapun. Regiunile ce corespund aceleiași zone din imagine sunt apoi concatenate pentru a obține reprezentarea. O modalitate asemănătoare de reprezentare a evenimentelor este folosită și de *Ionescu et al.* [?].

Sultani et al. [?] definește anomaliile la nivel de video, și tratează segmente din video drept evenimente în cadrul videoului. Acestea sunt analizate folosind un MIL (“multiple instance learning”) adânc. Această abordare este minim-supervizată, iar prin etichetarea la nivel de video, în locul etichetelor clasice, la nivel de cadru, a fost posibilă crearea și evaluarea pe un set de dimensiuni deosebit de mari, însumând 128 de ore de filmare.

Asemănător metodei alese spre implementare [?], au fost propuse multiple arhitecturi [?, ?] ce folosesc extragerea caracteristicilor unui eveniment în mod nesupervizat folosind autoencodere.

1.4 Conținutul lucrării

Ca și structură, lucrarea este împărțită în 2 părți:

- Partea algoritmică a sistemului de detectare a anomaliilor în video
- Partea de deployment a sistemului

În prima parte sistemul va fi analizat și detaliat din punct de vedere algoritmic și teoretic studiind problema detecției propriu zisă. Ca și tehnologii, în această parte am ales să folosesc Python3 ca și limbaj de programare pentru avantajele pe care le are. Printre acestea se numără faptul că este un limbaj orientat pe obiecte care pune la dispoziția dezvoltatorului numeroase librării specifice pentru AI/ML mutând astfel atenția dinspre detalii de implementare spre detalii de arhitectură și probleme mai abstracte ale programului care sunt cu adevărat importante pentru rezultatul final. Una dintre librăriile care s-a dovedit esențială dezvoltării este *Keras* [?], care oferă un API ce ușurează dezvoltarea unei rețele neuronale adânci/convoluționale dar și optimizează timpul de antrenare pentru modelele create.

În cea de a doua parte este analizat tipul de deployment al aplicației. Aici vor fi prezentate analize detaliate ale avantajelor și dezavantajelor soluțiilor cloud, dar și motivele pentru care modul final de deployment a fost ales.

Soluțiile de cloud-computing analizate vor fi:

- Infrastructure as a Service (IaaS) - analiză pe Amazon EC2
- Platform as a Service (PaaS) - analiză pe Amazon Elastic Beanstalk
- Function as a Service (FaaS) - analiză pe Amazon Lambda

Capitolul 2

Analiza arhitecturii si a tehnologiilor folosite

Pentru o mai bună înțelegere a contextului în care tehnologiile prezentate au fost folosite, acestea vor fi prezentate mai jos în cadrul secțiunii corespunzătoare locului în care a fost folosită în cadrul proiectului. Pentru toate etapele sistemului, limbajul folosit este Python3, deoarece, împreună cu librăriile și framework-urile existente pentru inteligență artificială, reprezintă mediul ideal pentru dezvoltarea unei soluții modulare, rapide și ușor de modificat.

2.1 Etapa de antrenare

Pentru etapa de antrenare, sistemul trece prin următoarea serie de etape:

- Detectează obiectele din setul de date
- Antrenează un autoencoder pe imaginile obiectelor și un alt autoencoder pe gradientii obiectelor.
- Obține reprezentarea latentă a fiecărui eveniment
- Stabilește k clase de normalitate folosind reprezentările latente
- Antrenează k clasificatori de tipul one-versus-rest

2.1.1 Autoencoder

Ca prim pas, pentru obținerea obiectelor dintr-o imagine, atât în etapa de antrenare cât și în cea de inferență este folosit un detector de obiecte pre-antrenat. Un detector de obiecte este un sistem ce primește ca input o imagine și după procesarea acesteia rezolvă problema detecției de obiecte și întoarce pozițiile la care sunt plasate obiectele în imagine, și etichetele asociate acestora. Deși detectorul de obiecte face parte din arhitectură, deoarece detectarea de obiecte este un subiect în sine, implementarea unui

astfel de algoritm nu face obiectul acestei lucrări. Datorită acestui lucru, este folosit un detector deja antrenat și testat pe setul de date COCO.

După aceasta etapă sunt antrenate cele două autoencodere convoluționale. Unul pentru imaginea propriu-zisă și altul pentru gradient. Un autoencoder este un tip de rețea neuronală alcătuit din 2 părți (encoder și decoder) care este folosit pentru a obține reprezentarea latentă a unui obiect folosind un mod de învățare nesupervizată. În timpul antrenării, autoencoder-urile au ca scop modificarea parametrilor interni pentru a obține la ieșire, datele primite la intrare. Deși ieșirea unui autoencoder nu prezintă interes decât în etapa de antrenare, ceea ce este folositor este reprezentarea latentă (rezultatul encoder-ului) a datelor. Folosind această tehnică, se obține o reducere a dimensionalității ce îmbunătățește semnificativ performanțele clasificatorilor ulteriori.

Operatorul de convoluție permite înțelegerea datelor de intrare prin aplicarea unui kernel, ce extrage informația prin combinarea datelor asupra cărora este aplicat, fapt ilustrat în figura 2.1. Autoencoderele tradiționale nu iau în considerare faptul că un semnal poate fi văzut ca o sumă de alte semnale. Operatorul de convoluție a fost introdus în acestea tocmai pentru a exploata această posibilitate. Autoencoderele convoluționale reprezintă cea mai bună unealtă pentru învățarea nesupravegheată a filtrelor convoluționale. Spre deosebire de cazurile în care filtrele convoluționale sunt construite manual pentru a extrage cât mai bine anumite trăsături specifice din imagine, autoencoderele convoluționale învață(construiesc) în mod automat filtrele în perioada de învățare, având ca scop minimizarea erorii de reconstrucție. Acestea reprezintă metoda perfectă pentru a obține o reprezentare compactă a unor date abstracte și multi dimensionale, cum sunt imaginile digitale.[?]

Obținerea reprezentării latente a fiecărui obiect se realizează trecând imaginea, respectiv gradientul său prin encoderul autoencoderului corespunzător și păstrarea rezultatului.

2.1.2 Clusterizare K-means

Odată obținut vectorul de caracteristici pentru toate obiectele din setul de date, urmează stabilirea claselor de normalitate. Acest lucru se realizează prin aplicarea algoritmului k-means de clustering. Pentru implementare a fost folosit algoritmul *LLoyd* implementat în biblioteca python *sklearn*. Aplicând acest algoritm peste vectorii

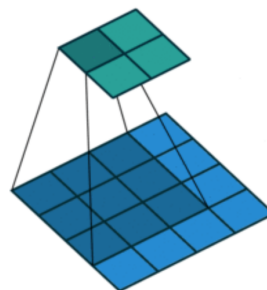


Figura 2.1: Aplicarea operatorului de convoluție

Sursă: *A guide to convolution arithmetic for deep learning* [?]

de caracteristici obținute, rezultă k categorii de normalitate cu vectorii de caracteristici aferenți.

K-means este un tip de clusterizare ce are ca scop împărțirea a n probe în k grupe. Pentru ca grupele create să reprezinte clase de obiecte cât mai apropiate, algoritmul are ca scop minimizarea pătratului distanței Euclidiene dintre probe. Algoritmul *LLoyd* rezolvă această problema prin execuția unei inițializări (alegerea aleatorie a k centre) și repetarea a doi pași principali, până când problema converge la un minim local. Pașii executați sunt :

- Atribuire fiecare element unei grupe. Elementul este atribuit grupei față de care criteriul de selecție (pătratul distanței Euclidiene față de centrul grupei) este minim.
- Recalculează noi centre pentru fiecare grupa, pe baza noilor elemente. Centrele sunt calculate făcând media elementelor din fiecare grupă.

2.1.3 Clasificare multi-class folosind SVM

Folosind categoriile de normalitate obținute la pasul anterior, putem spune că față de o anumită categorie i , celelalte $k-1$ categorii reprezintă categorii *artificial* anormale. Le numim *artificial* anormale deoarece în mod obiectiv ele sunt acțiuni normale pentru sistemul de detectare a anomaliilor, dar pentru antrenarea unor clasificatori conform schemei one-vs-rest acestea sunt tratate drept anormale. Astfel, putem antrena un clasificator binar $g(i)$ în așa fel încât să separăm elementele din categoria i de cele din categoriile $\{1,2,...k\}/i$ generând funcția :

$$f_i(x) = \sum_{j=1}^n w_j * x_j + b$$

, unde $x \in R^n$ reprezintă vectorul de caracteristici, w este vectorul de parametri a funcției, iar b reprezintă bias-ul funcției. [?].

Astfel, generăm k astfel de funcții corespunzătoare celor k clasificatori ce vor fi folosiți pentru a stabili dacă un eveniment este anormal. Conform schemei one-vs-rest, un eveniment este anormal dacă este clasificat drept anormal de către toți cei k clasificatori.

Funcția f_i reprezintă un SVM (mașină cu vectori suport) ce are ca scop clasificarea unor probe în diferite clase. În timpul perioadei de antrenare un SVM construiește un hiperplan ce separă cât mai precis datele de antrenare în 2 clase conform etichetelor acestora, maximizând distanța minimă față de planul de decizie. Astfel, scopul unei mașini cu vectori suport este găsirea unui \vec{w} minim (norma acestuia să fie minimă) astfel încât: $y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1$, pentru toți $1 \leq i \leq n$, unde y_i reprezintă eticheta probei x_i . Ce înseamnă ca toate probele etichetate cu 1 se află de o parte a planului de decizie, și toate probele etichetate cu -1 se află de cealaltă parte.

2.2 Etapa de inferență

În cadrul etapei de inferență, sistemul folosește detectorul de obiecte, autoencoderele pre-antrenate și cei k clasificatori binari pentru a stabili dacă un anumit eveniment este sau nu anormal. Astfel, parcursul sistemului este următorul :

- Extragerea cadrelor necesare din video
- Extragerea obiectelor din imagini
- Obținerea reprezentării latente
- Clasificarea evenimentelor

Pentru analiza unui eveniment care apare la un indice dat t sunt necesare 3 cadre. Mai precis cadrele de la indicii $t-3$, $t+3$ și t . Din cadrul t se va extrage vectorul de caracteristici specific aparenței vizuale, iar din celelalte 2 cadre se vor extrage vectorii de caracteristici specifici mișcării obiectului, prin analiza gradientilor. Prin concatenarea acestor 3 vectori, se obține vectorul final de caracteristici ce va fi folosit drept input pentru clasificatorii finali.

Pentru extragerea obiectelor din cadrele analizate, se va folosi același detector de obiecte ca în etapa de antrenare. Acesta va fi rulat pe cadrul principal t , urmând apoi să se folosească coordonatele obiectelor de la cadrul t , și pentru cadrele $t+3$ și $t-3$ deoarece din cauza diferenței mici de indici, obiectele nu se pot mișca îndeajuns încât să fie necesară rularea pe toate cele 3 cadre.

Odată obținute obiectele din cadrul analizat, după obținerea gradientilor din cadrele $t-3$ și $t+3$, se poate obține reprezentarea latentă a acestor informații. Reprezentarea latentă a imaginii obiectului constă în rezultatul generat de encoderul autoencoderului pentru imagini iar reprezentarea latentă a gradientilor constă în rezultatul generat de encoderul autoencoderului pentru gradienti.

Odată obținuți vectorii de caracteristici pentru reprezentarea vizuală și pentru reprezentarea mișcării obiectului, prin concatenarea lor se obține vectorul final, ce poate fi folosit drept input pentru clasificarea finală. Conform schemei *one-vs-rest* acest vector este clasificat de toți cei k clasificatori, iar rezultatul final este scorul maxim cu semn schimbat obținut în urma clasificării.

2.3 Etapa de lansare în cloud

Pentru a face sistemul public, acesta este lansat drept un API ce rulează etapa de inferență pe un server web plasat în cloud. Pentru dezvoltarea serverului am ales să folosesc framework-ul *Flask*. Flask este un micro-framework de python folosit pentru dezvoltarea soluțiilor web. Motivele pentru care acest framework a fost ales sunt :

- Acesta adaugă un număr de dependențe suplimentare foarte mic aplicației, lucru esențial atunci când aplicația se dorește a fi plasată în cloud, din cauza limitărilor de memorie.
- Oferă un suport foarte bun pentru planificarea rutelor de intrare în aplicație, lucru foarte important atunci când se dorește dezvoltarea unui API.
- Este unul dintre framework-urile suportate de Amazon Elastic Beanstalk

Comparativ cu alte framework-uri, Flask este diferit deoarece nu impune linii clare dezvoltatorilor atunci când vine vorba de forma sau componentele aplicației ce urmează a fi dezvoltată. Astfel, dezvoltatorul are control complet asupra aplicației și își poate manifesta creativitatea sau ideile fără a fi restricționat de framework. Flask a fost creat tocmai cu ideea de a fi construit peste el. Deși poate nu oferă aceeași viteză de dezvoltare comparativ cu celelalte frameworkuri, acesta oferă libertatea de alegere la fiecare pas. Are suport pentru toate tipurile de baze de date, fie ele relaționale sau nerelaționale, nu are preferințe când vine vorba de metode de autentificare sau de creare a rolurilor, totul este suportat și totul este la latitudinea dezvoltatorului. [?]

Pentru a lansa serverul în cloud, am folosit serviciul Amazon Elastic Beanstalk. Acesta este un serviciu complex, ce însumează la rândul lui mai multe servicii cloud oferite de Amazon Web Services(AWS). Mai jos sunt prezentate câteva dintre avantajele și dezavantajele acestui serviciu, fiind analizat în detaliu în capitolele ce urmează. Elastic Beanstalk este un serviciu de tipul *PaaS* ce oferă servicii de deployment și administrare complete. Pentru o mai bună detaliere, mai jos sunt definite toate serviciile cloud incluse de Elastic Beanstalk:

- Amazon EC2 : este un serviciu web oferit de Amazon ce constă în oferirea unui mediu de execuție sigur în cloud. Este echivalentul unei mașini fizice, doar că este accesată prin intermediul internetului. Acesta a fost creat pentru a ușura misiunea dezvoltatorilor de a migra serviciile proprii spre cloud computing. Aceste instanțe sunt extrem de configurabile, punând la dispoziția dezvoltatorului mai mult de 50 de tipuri de instanțe, plus diferite opțiuni de optimizare a unor părți specifice, cum ar fi memoria sau placa video. [?]
- Amazon S3 : este un serviciu ce oferă medii de stocare în cloud. Stocarea este de tipul cheie-obiect, unde cheia identifică unic la nivel global un fișier. Acesta oferă o securitate sporită a datelor, și o disponibilitate de 99.9999999% deoarece datele sunt distribuite în sisteme diferite aflate în zone diferite. [?]
- Auto Scaling Group : acest serviciu oferă un mod automat de lansare a instanțelor EC2 astfel încât traficul să nu depășească niciodată puterea de execuție a unui aplicații. Scopul acestui serviciu este de a oferi capacitatea de a scala pentru a menține o performanță optimă, menținând costul în tot acest timp la valoarea minimă. [?]

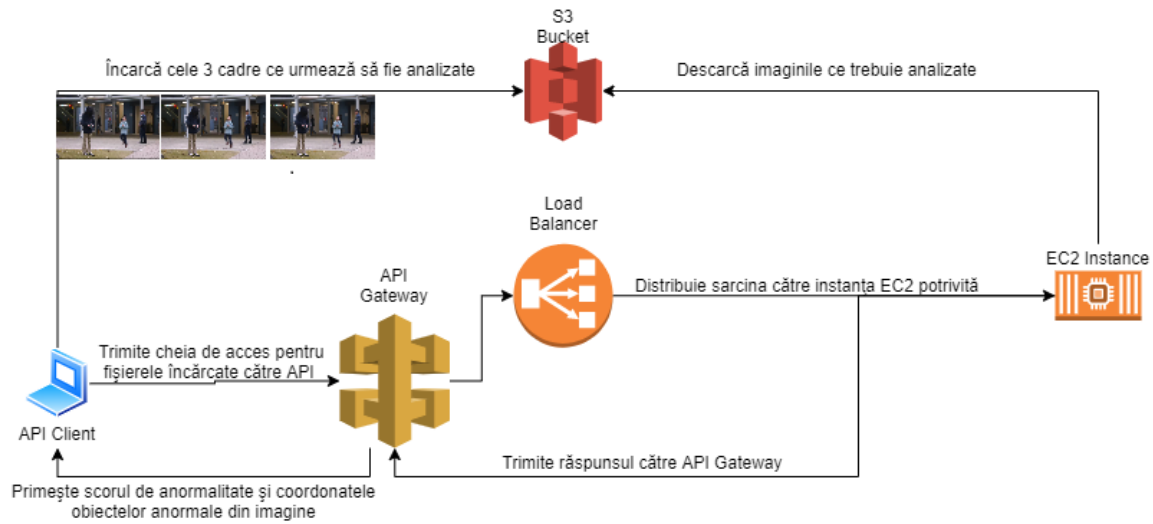


Figura 2.2: Arhitectura abstractizată a API-ului

- Elastic Load Balancing: este un serviciu care împarte traficul administrat de aplicație către instanțele lansate astfel încât acestea să fie utilizate într-un mod optim. Astfel, poate suporta încărcătura variabilă a aplicației și o poate distribui în așa fel încât, în funcție de modul ales (*accesibilitate crescută*, (*scalare automată*), *securitate ridicată*) aplicația să nu scadă sub performanțele dorite. [?]

Dezvoltatorul se ocupă doar de aplicația/serverul propriu zis, crează pachetul de deployment, iar Elastic Beanstalk crează instanțele EC2 necesare, administrează și rutează traficul către instanțe folosind un Load Balancer, iar atunci când aplicația este suprasolicitată, lansează în mod automat noi servere folosindu-se de avantajele unui Auto Scaling Group.

Așa cum se poate observa în figura 2.2, API-ul este creat în așa fel încât evaluarea se face pentru fiecare cadru în parte. Dezvoltatorul ce implementează clientul pentru API are doar responsabilitatea urcării cadrelor necesare într-un spațiu S3 prestabilit. Preprocesarea, extragerea obiectelor din imagine, și rularea detecției de anomalii sunt toate executate în cloud. Astfel, efortul computațional asupra clientului este minim. La momentul accesării API-ului, serverul trebuie să primească ca parametru în apelul HTTP cheia de acces pentru cadrele urcate de către client. Folosind această informație, pe server se descarcă aceste cadre și sunt analizate pentru a detecta anomaliile din cadrul central. Ca și rezultat, clientul primește scorul de anormalitate al cadrului împreună cu toate pozițiile obiectelor anormale din cadru, date ce pot fi folosite pentru notificări ulterioare sau diferite aplicații pe partea de client. Un avantaj important al unui HTTP API este abilitatea de a fi accesat prin internet, interfața API-ului fiind accesibilă pentru toți utilizatorii fără să necesite instrucțiuni speciale, precum *SOAP* ce este bazat pe *XML*.

2.4 Tehnologii folosite în dezvoltarea interfeței

Pentru a putea vizualiza capacitățile sistemului de detecție a anomaliilor, am dezvoltat un program ce simulează cele 2 moduri în care poate fi acest sistem folosit : prin rulare locală, sau prin execuție în cloud.

Pentru a crea interfața grafică, am ales să folosesc framework-ul *Electron*. Electron este un framework creat pentru a facilita dezvoltarea de aplicații desktop cross-platform, folosind tehnologii web. Acesta este bazat pe *Node.js* și *Chromium* și pune la dispoziția dezvoltatorului folosirea tehnologiilor web precum:

- HTML
- CSS
- JavaScript

Datorită posibilității de a folosi tehnologii web pentru a dezvolta interfața unei aplicații desktop, se pot crea interfețe atractive, stilizate și aranjate astfel încât experiența utilizării programului să fie cât mai bună.

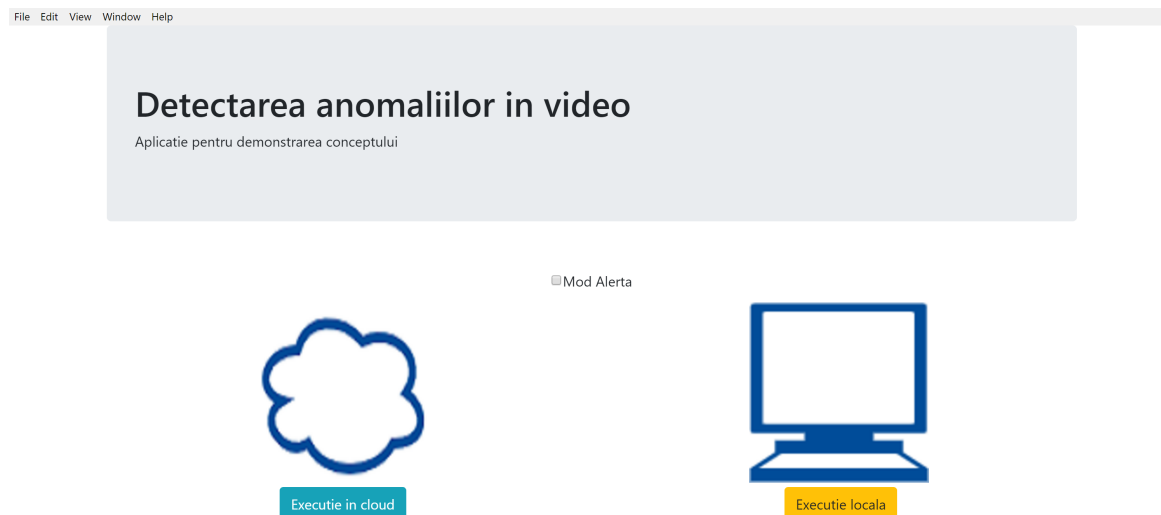


Figura 2.3: Pagina de start a aplicației

Programul creat simulează un client ce folosește sistemul de detecție a anomaliilor și anume : rulează detecția de anomalii (local sau în cloud, în funcție de metoda selectată) folosind o cameră video atașată sistemului de pe care este rulat. Modul în care sunt evidențiate evenimentele anormale este deasemenea configurabil :

- Modul standard : Încadrarea în cadrane verzi a obiectelor ce sunt detectate de către sistem ca având un comportament anormal.
- Modul alertă : Programul extrage obiectul din imagine atunci cand acesta este detectat ca fiind anormal, și creează o alertă conținând doar această imagine pe care o afișează pe ecran. Acest mod de alertă simulează scenariul în care este necesară atragerea atenției în mod suplimentar a operatorului în momentul apariției unei anomalii.

Capitolul 3

Sistemul de detecție a anomaliilor

În acest capitol va fi analizat în detaliu sistemul de detecție a anomaliilor în video, atât din punct de vedere al arhitecturii alese cât și din punct de vedere al implementării.

3.1 Detecția obiectelor

În ceea ce privește detecția de obiecte, aceasta este comună tuturor etapelor sistemului. Așa cum se poate observa și în secțiunea de analiză a performanțelor sistemului, detecția obiectelor din imagine joacă un rol esențial în viteza finală de analiză.

Prima opțiune a fost un detector de obiecte pre-antrenat pe setul de date COCO ce folosește o arhitectură **SSD-MobileNet**, din biblioteca *gluoncv model-zoo*. Acesta are ca prim avantaj viteza de analiză pe GPU a unei imagini, dar și precizia bună a detecțiilor. Fiind asemănător cu cel folosit de *Ionescu et al.*, acesta folosește o arhitectură ce poate fi exploatată la potențialul ei maxim doar pe o mașină ce posedă un GPU puternic, având o viteza insuficientă pe sisteme bazate doar pe CPU.

Deoarece sistemul actual are ca scop final rularea pe o platforma ce necesită resurse minime, atât în cazul rulării offline cât și online, am ales folosirea unui model cu o arhitectură optimizată pentru rularea pe CPU. Mai precis, un model cu o arhitectură **YOLOv3-tiny** preluat din biblioteca *libcv*. Această schimbare a adus o îmbunătățire de **10x** în ceea ce privește timpul de detecție a obiectelor per cadru, de la o medie de 1200 de ms la 120 ms.

Detectorul YOLO este un tip de detector cu o singură etapă, ceea ce semnifică că nu mai este executată etapa de propunere a regiunilor întâlnite în detectoarele cu 2 etape, precum *R-CNN* sau *Fast R-CNN*. Astfel, imaginea este împărțită în $S \times S$ pătrate. Fiecare pătrat este responsabil să detecteze obiectul aflat în interiorul acestuia. Fiecare pătrat are atribuit un scor ce arată probabilitatea ca un obiect să se afle în acesta, și încă un scor, ce reprezintă intersecția supra reuniune dintre perimetrul prezis și poziția adevărată a obiectului din imagine. Pentru fiecare astfel de pătrat sunt propuse *bounding box-uri* și scorul de încredere aferent. Un astfel de detector are la bază nivele convoluționale ce extrag caracteristici din imagine urmată de nivele conectate

complet pentru clasificare. Desigur, YOLOv3-tiny a suferit numeroase modificări în scopul optimizării arhitecturii din punct de vedere al timpului de execuție.[?] Odată cu viteza apare în schimb un dezavantaj în ceea ce privește detecția de obiecte: datorită arhitecturii minimaliste, detectorul ales detectează obiectele în integritatea lor doar dacă sunt relativ aproape de cameră și nu sunt foarte mici.

Implementarea detecției de obiecte a fost realizată la nivelul sistemului în cadrul clasei *ObjectDetector* ce prezintă următoarele funcționalități :

- La momentul instanțierii rulează detectorul de obiecte pre-antrenat pe imaginea primită ca parametru, obținând astfel cadranele delimitatoare ale obiectelor din imagine împreună cu scorurile și clasele aferente.
- Oferă prin intermediul funcției *get_object_detections* obiectele extrase din imagine pe baza marginilor delimitatoare obținute în timpul instanțierii. Obiectele sunt redimensionate la 64×64 și returnate sub forma unui vector de imagini.
- Oferă prin intermediul funcției *get_detections_and_cropped_sections* ce primește ca parametru și imaginea de la indexul $t-3$ și imaginea de la indexul $t + 3$ relativ la indexul t al imaginii folosite pentru instanțiere pachetul complet pentru trecerea prin sistemul de detecție și anume : Folosind marginile delimitatoare obținute în timpul instanțierii, sunt extrase obiectele din toate cele 3 imagini, și returnate sub forma a 3 vectori de imagini. Astfel un eveniment va fi format din cele 3 imagini aflate la un același index i în cei 3 vectori.

3.2 Analiza antrenării sistemului

Din cauza complexității problemei de a identifica comportamentul anormal al obiectelor prezente în video, arhitectura sistemului presupune multiple etape de prelucrare a datelor până la momentul clasificării finale. Astfel, etapa de antrenare este împărțită la rândul ei în 2 etape:

- Etapa de reducere a dimensionalității (antrenarea autoencoderelor)
- Etapa de antrenare a clasificatorilor finali

În etapa de reducere a dimensionalității sunt definite și antrenate autoencoderile pe baza imaginilor și gradientilor extrași din setul de date. Motivul pentru care obiectele sunt procesate de către autoencodere este că datorită antrenării doar pe obiectele din videourile de antrenare, acestea vor învăța să reprezinte mai bine evenimentele normale. Astfel, atunci când prin aceste autoencodere vor trece evenimente anormale, ce nu sunt asemănătoare cu cele de antrenament, autoencoderile vor genera o eroare de reconstrucție ce va ușura sarcina clasificatorilor finali.

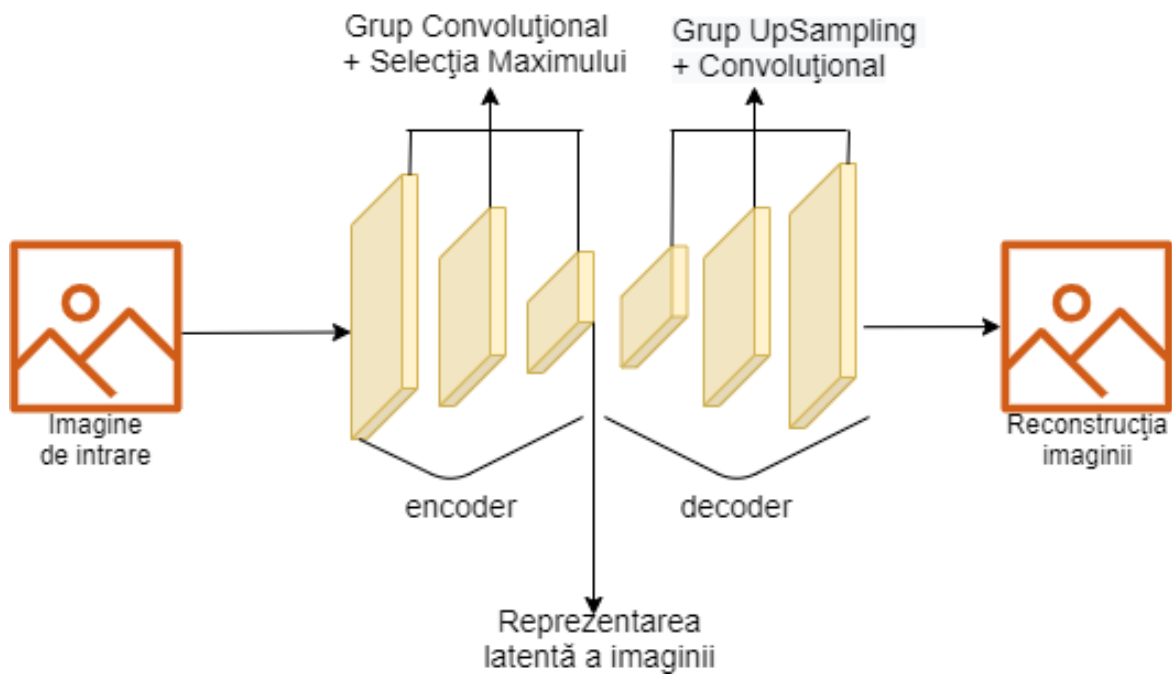


Figura 3.1: Prezentarea principiului de funcționare a autoencoderelor și arhitectura lor generală

Arhitectura aleasă pentru autoencodere este cea descrisă de *Ionescu et al.* [?] și constă într-o arhitectură convoluțională rapidă, formată dintr-un encoder cu 3 blocuri convoluțional + max-pooling și un decoder format din 3 blocuri upsampling + convoluțional. Fiecare autoencoder primește ca input date de dimensiune 64×64 și creează după encoder un vector de caracteristici de dimensiune $8 \times 8 \times 16$. Structura detaliată a autoencoderelor este :

- Stratul de intrare, de dimensiune $64 \times 64 \times 1$
- Bloc Convoluțional + MaxPooling format din : Un strat convoluțional bazat pe 32 de filtre de dimensiune 3×3 urmat de funcția de activare *Relu* și un strat max-pooling bazat pe filtre 2×2 cu *stride* 2.
- Bloc Convoluțional + MaxPooling format din : Un strat convoluțional bazat pe 32 de filtre de dimensiune 3×3 urmat de funcția de activare *Relu* și un strat max-pooling bazat pe filtre 2×2 cu *stride* 2.
- Bloc Convoluțional + MaxPooling format din : Un strat convoluțional bazat pe 16 de filtre de dimensiune 3×3 urmat de funcția de activare *Relu* și un strat max-pooling bazat pe filtre 2×2 cu *stride* 2.
- Bloc Convoluțional + UpSampling format din : Un strat convoluțional bazat pe 16 de filtre de dimensiune 3×3 urmat de funcția de activare *Relu* și un strat UpSampling bazat pe filtre 2×2 .

- Bloc Convoluțional + UpSampling format din : Un strat convoluțional bazat pe 32 de filtre de dimensiune 3×3 urmat de funcția de activare *Relu* și un strat UpSampling bazat pe filtre 2×2 .
- Bloc Convoluțional + UpSampling format din : Un strat convoluțional bazat pe 32 de filtre de dimensiune 3×3 urmat de funcția de activare *Relu* și un strat UpSampling bazat pe filtre 2×2 .
- Un ultim strat Convoluțional ce are ca rol reducerea dimensiunii de ieșire de la $64 \times 64 \times 32$ la $64 \times 64 \times 1$ [?] . Acesta este bazat pe 1 filtru de dimensiune 3×3 urmat de funcția de activare *Sigmoid*

Primele 4 straturi reprezintă encoder-ul, în timp ce ultimele 4 straturi reprezintă decoderul.

În cadrul sistemului, clasa *AutoEncoderModel* implementează funcționalitățile necesare și anume :

- La momentul instanțierii definește arhitectura autoencoderului.
- Prin intermediul metodei *__train_autoencoder* antrenează autoencoderul pe baza unui vector de imagini primit ca parametru. Înainte de a începe antrenarea propriu zisă se verifică dacă există deja un model pre-antrenat în sistemul de fișiere. Dacă nu există, se realizează antrenarea completă, urmată de salvarea modelului în sistemul de fișiere.
- Prin intermediul metodei *get_encoded_state* ce primește ca parametru o imagine, returnează reprezentarea latentă a acelei imagini.

Implementarea clasei **AutoEncoderModel** este :

```
class AutoEncoderModel:
    """
    _____
    autoencoder = Modelul pentru autoencoderul complet,
    contine atat encoderul cat si decoderul
    encoder = doar encoderul, extras din autoencoder, avand aceleasi greutate.
    """
    def __init__(self, input_images, checkpoints_name):
        self.checkpoints_name = checkpoints_name
        #fișierul in care vor fi salvate modelele dupa antrenare
        self.checkpoint_dir = '/home/some_directory_%s' % self.checkpoints_name
        self.num_epochs = 100
        self.batch_size = 64
        self.autoencoder, self.encoder = self.__generate_autoencoder()
        self.__train_autoencoder(input_images)

    def __generate_autoencoder(self):
        input_img = Input(shape=(64, 64, 1))
```

```

x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), strides=2, padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), strides=2, padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), strides=2, padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
autoencoder = Model(input_img, decoded)
encoder = Model(input_img, encoded)
# compileaza modelele folosind optimizatorul Adam si
# diferenta medie patratica drept functie loss
optimizer = Adam(lr=10 ** -3)
encoder.compile(optimizer=optimizer, loss='mse')
autoencoder.compile(optimizer=optimizer, loss='mse')
print(autoencoder.summary())
return autoencoder, encoder

def _train_autoencoder(self, input_images):
    """
    Parametri
    """
    input_images = np.array ce contine imagini de dimensiune 64x64x1
    ce vor fi folosite pentru a antrena autoencoderul
    """
    if not os.path.exists(self.checkpoint_dir):
        os.makedirs(self.checkpoint_dir)
        checkpoint_callback =
            ModelCheckpoint(filepath=self.checkpoint_dir + '/weights.hdf5',
                            verbose=1,
                            save_best_only=True)
        early_stopping_monitor = EarlyStopping(patience=2)
        data_train, data_test, gt_train, gt_test =
            train_test_split(input_images, input_images, test_size=0.20,
                            random_state=42)
        self.autoencoder.fit(data_train, data_train,
                            epochs=self.num_epochs,
                            batch_size=self.batch_size,
                            validation_data=(data_test, data_test),
                            callbacks=[checkpoint_callback, early_stopping_monitor])
    else:
        self.autoencoder.load_weights(self.checkpoint_dir + '/weights.hdf5')

```

```
def get_encoded_state(self, image):
    """
    Parametri
    -----
    images – np.array ce contine imaginea ce trebuie codificata
    Rezultat
    -----
    np.array ce contine imaginea codificata de catre encoder
    """
    input = np.expand_dims(image, axis = 0)
    encodings = self.encoder.predict(input)
    return encodings[0]
```

[?]

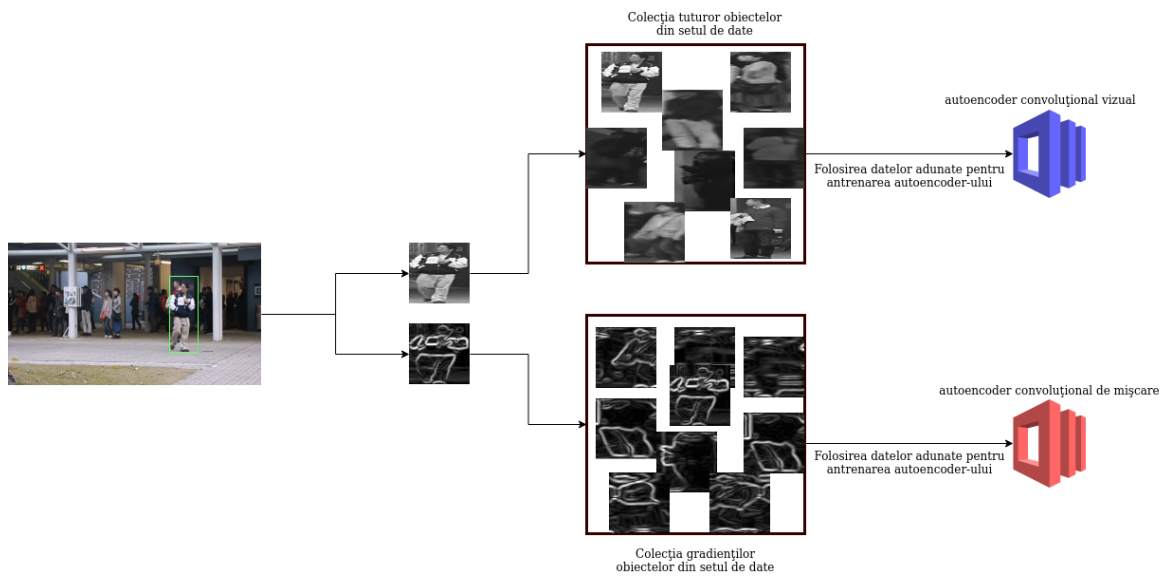


Figura 3.2: Arhitectura primei etape de antrenare

Implementarea arhitecturii prezentate în figura 3.2 constă în următorii pași:

- Pentru toate videourile de antrenare se execută detecția de obiecte pentru fiecare cadru, extrăgând astfel toate obiectele din video. Fiecare cadru este transformat în alb-negru pentru a fi prelucrat în etapele următoare. Pentru fiecare obiect extras, imaginea acestuia este redimensionată pentru a respecta dimensiunea de intrare a autoencodere-lor la 64×64 iar apoi este calculat gradientul, ce reflectă mișcarea obiectului. Gradientul este calculat după formula:

$$\sqrt{G_x^2 + G_y^2}$$

Unde G_x , G_y sunt imagini ce în fiecare punct conțin derivata orizontală respectiv verticală a imaginii inițiale, obținută prin aplicarea unui kernel Sobel de 3×3 .

Dimensiunea kernelului folosit pentru calcularea gradientilor este foarte importantă pentru extragerea precisă a detaliilor, deoarece, așa cum se poate observa în

figura 3.3, cu cât kernelul este mai mare, cu atât trasaturile extrase sunt mai generale.

- Imaginile și gradientii astfel obținuți sunt adaugați într-o colecție generală pentru tot setul de date. Cele 2 colecții astfel create vor servi drept input pentru antrenarea autoencoderelor.
- Folosind cele 2 colecții (de imagini, respectiv de gradienti) este antrenat câte un autoencoder pentru fiecare colecție. Înainte de a fi folosite pentru antrenarea autoencoderelor, atât imaginile, cât și gradientii, sunt normalizate în intervalul $[0,1]$. Antrenarea se realizează folosind optimizatorul Adam [?] și funcția loss ce constă în diferența medie a pătratelor, dată de formula : $L(I, O) = \frac{1}{h*w} \sum_1^h \sum_1^w (I_{ij} - O_{ij})^2$ [?] unde I și O reprezintă imaginea de input respectiv de output și h,w sunt dimensiunile imaginilor. Aceasta este executată timp de 100 de epoci cu o rată de învățare de 10^{-3} , dar având și o regula de oprire rapidă, ce înseamnă că dacă timp de 2 epoci funcția loss nu se îmbunătățește, antrenarea este finalizată.

În etapa de antrenare a clasificatorilor finali, se folosesc autoencoderule antrenate în etapa precedentă pentru a obține vectorii de caracteristici specifici fiecărui eveniment prezent în setul de date. Ca și prim pas, se detectează toate obiectele prezente în fiecare cadru din video-urile de antrenare, iar pentru fiecare obiect, se folosesc coordonatele acestuia pentru a decupa același obiect din cadrul $t-3$ și $t+3$, respectiv la indexul t curent. Se calculează gradientii pentru imaginile decupate, astfel este reprezentată mișcarea obiectului față de cadrul curent, iar apoi se obțin vectorii caracteristici ai fiecarui gradient prin trecerea acestora prin autoencoderul corespunzător antrenat în etapa precedentă. Odată obținuți cei 3 vectori caracteristici specifici evenimentului, așa cum este ilustrat și în figura 3.4, concatenarea acestora este stocată într-o colecție globală ce reține datele pentru tot setul de date. Obținerea vectorului de caracterisitici pentru un anumit eveniment este realizată în cadrul funcției :

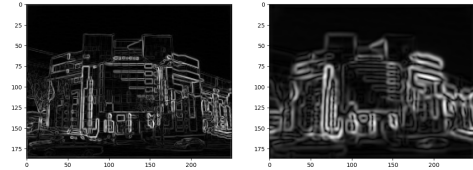


Figura 3.3: Gradienti obținuți în urma aplicării unui kernel 3×3 , respectiv unul de dimensiune 21×21

```
def _get_feature_vectors_and_bboxes(self, frame, frame_d3, frame_p3):
    """
    :param frame: np.array - Cadrul ce trebuie analizat
    :param frame_d3 : np.array - Cadrul de la indexul t-3 comparativ
    cu indexul t al cadrul ce trebuie analizat. d3 vine de la delta3
    :param frame_p3 : np.array - Cadrul de la indexul t+3 comparativ
    cu indexul t al cadrul ce trebuie analizat. p3 vine de la plus3
    """
    object_detector = ObjectDetector(frame)
    bounding_boxes = object_detector.bounding_boxes
    cropped_detections, cropped_d3, cropped_p3 =
```

```

object_detector.get_detections_and_cropped_sections(frame_d3, frame_p3)
gradient_calculator = GradientCalculator()
gradients_d3 =
    self._prepare_data_for_CNN(gradient_calculator
                               .calculate_gradient_bulk(cropped_d3))
gradients_p3 =
    self._prepare_data_for_CNN(gradient_calculator
                               .calculate_gradient_bulk(cropped_p3))
cropped_detections = self._prepare_data_for_CNN(cropped_detections)
list_of_feature_vectors = []
for i in range(cropped_detections.shape[0]):
    apperance_features =
        self.trainer_stage2.autoencoder_images
            .get_encoded_state(np.resize(cropped_detections[i]
                                         , (64, 64, 1)))
    motion_features_d3 = self.trainer_stage2.autoencoder_gradients
        .get_encoded_state(np.resize(gradients_d3[i]
                                     , (64, 64, 1)))
    motion_features_p3 = self.trainer_stage2.autoencoder_gradients
        .get_encoded_state(np.resize(gradients_p3[i]
                                     , (64, 64, 1)))

    feature_vector =
        np.concatenate((motion_features_d3.flatten()
                        , apperance_features.flatten()
                        , motion_features_p3.flatten()))
    list_of_feature_vectors.append(feature_vector)
return np.array(list_of_feature_vectors), bounding-boxes

```

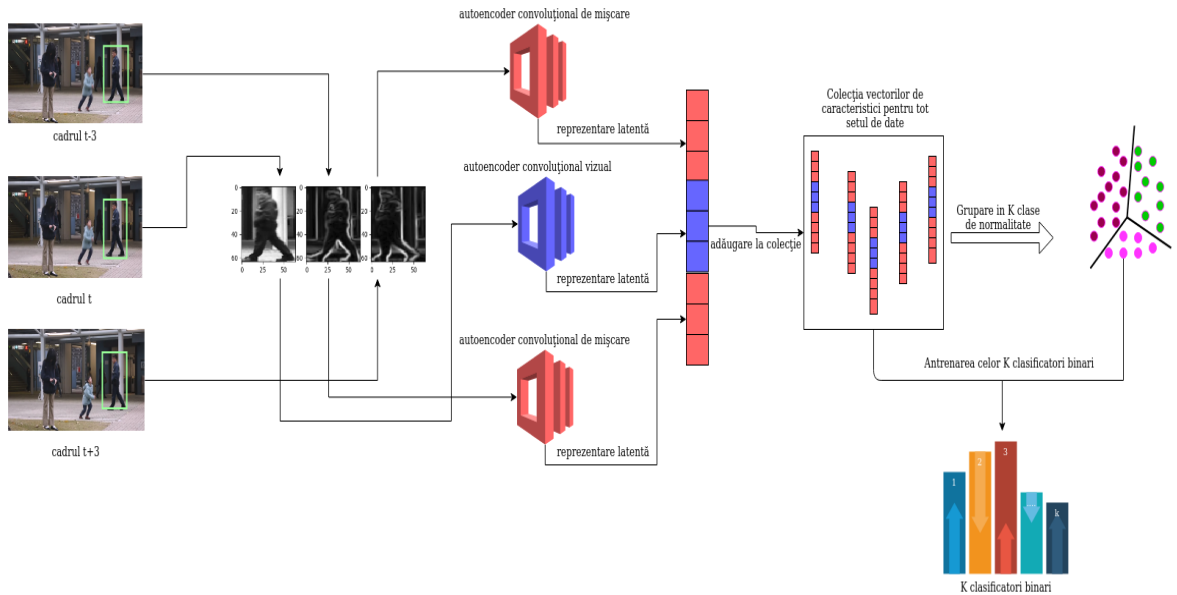


Figura 3.4: Arhitectura etapei finale de antrenare

Aplicând algoritmul de *k-means clustering* se obțin k categorii de normalitate. Astfel, pentru fiecare eveniment din colecția globală este cunoscută categoria i din

care face parte. Folosind aceste date, putem antrena cei k clasificatori binari. Pentru fiecare categorie, un clasificator binar este antrenat folosind evenimentele din categoria curent drept date de antrenare pozitive, iar celelalte $k-1$ categorii drept date de antrenare negative. În final, se obțin k clasificatori binari ce au rolul de a evalua dacă un eveniment aparține sau nu categoriei de normalitate i unde i este indicele clasificatorului g_i .

Ca și implementare, antrenarea sistemului a fost dezvoltată având ca scop crearea unui flux de lucru automat și configurabil. Astfel, pentru a reantrena sistemul folosind un alt set de date, este necesară doar schimbarea parametrului ce conține calea către video-urile de antrenament.

3.3 Analiza etapei de inferență

În cadrul etapei de inferență este analizat un eveniment cu scopul de a determina dacă acesta este sau nu anormal. Analiza unui singur eveniment stă la baza tuturor modurilor de utilizare a sistemului, deoarece, având aceste date, putem obține scoruri de normalitate pentru fiecare cadru, sau scoruri de normalitate la nivel de pixel sau chiar și clasificări generale de normalitate a unui video în totalitate.

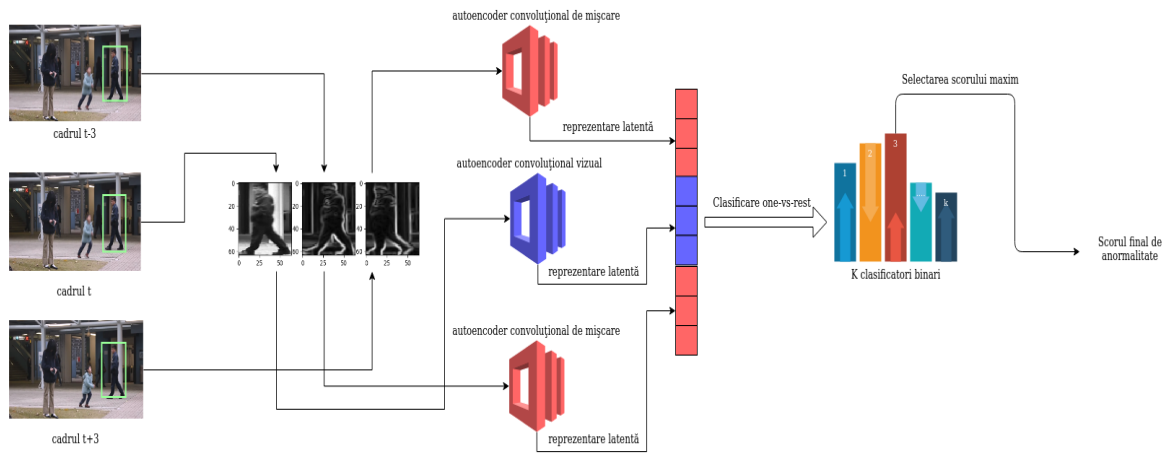


Figura 3.5: Arhitectura etapei de inferență

Pentru a obține scorul unui eveniment, un pas foarte important este obținerea vectorului de caracteristici corespunzător. Acesta se obține folosind autoencoderele antrenate în etapa precedentă, după pre-procesarea cadrelor corespunzătoare evenimentului. Pre-procesarea este asemănătoare etapei de antrenare, și anume, pentru fiecare obiect dintr-un cadru t , se selectează aceeași zonă din cadrele $t-3$ și $t+3$ cu scopul de a calcula gradientii ce reflectă mișcarea obiectului relativ la cadrul t . Odată obținute aceste 3 date (imaginea obiectului și cei 2 gradienti) așa cum este descris și în figura 3.5, acestea sunt trecute prin autoencodere și sunt obținute reprezentările latente, ce sunt concatenate pentru a se obține vectorul final de caracteristici. Vectorul de

caracteristici este apoi clasificat de toți cei k clasificatori binari antrenați anterior, iar scorul final de anormalitate este dat de formula :

$$score = -\max(g_i(v)), i \in [1..k]$$

unde v este vectorul de caracteristici, iar g_i este un clasificator binar. Implementarea funcției ce generează scorul de anomalie a unui eveniment este :

```
def get_inference_score(self, feature_vector):
    scores = self.model.decision_function([feature_vector])[0]
    return -max(scores)
```

Filmarea propriu zisă și selecția cadrelor este realizată pe partea de client, în timp ce pre-procesarea cadrelor, detecția obiectelor și calcularea scorului pentru fiecare dintre aceste obiecte este executat pe server. Astfel, atât timp cât există o conexiune la rețea, clientul nu este limitat de puterea de calcul proprie, și poate analiza mai multe videoclipuri simultan. Totuși clientul trebuie să încarce cadrele ce vor fi analizate către o destinație S3 a serverului. Pentru a nu încetini semnificativ procesul de analiză, trimitere cadrelor în rețea trebuie făcută în paralel, folosind eventualele posibilități de multi-threading ale clientului.

În ceea ce privește serverul pe care rulează analiza propriu zisă, acesta este un HTTP API, ce expune pentru public capacitatea de a rula inferența. În stadiul curent, API-ul este configurat cu un singur endpoint, de tip POST : “upload/frame_key” , unde *frame_key* reprezintă cheia de acces cu care au fost urcate în cloud cadrele ce urmează să fie analizate. Pentru ca o cheie de acces să fie considerată validă de către server, acesta trebuie să găsească în destinația S3 specifică 3 fișiere ce au următoarele chei :

- *frame_key*
- *frame_key_d3*
- *frame_key_p3*

Frame_key este parametrul primit prin HTTP, iar *frame_key_d3* și *frame_key_p3* sunt chei obținute prin concatenarea sufixului **_d3** respectiv **_p3** la cheia primită ca parametru. Dacă parametrul primit reprezintă o cheie validă, atunci cadrele sunt descărcate temporar pe server, unde sunt folosite pentru a rula întregul proces de inferență, iar în final, rezultatele sunt transmise clientului în format json. JSON-ul rezultat conține 3 câmpuri :

- *Codul de stare*: ce reprezintă codul http rezultat în urma apelului
- *Result*: ce conține scorul cadrului de anormalitate, ce reprezintă maximumul dintre scorurile obiectelor din cadru.
- *boxes*: o listă ce conține coordonatele obiectelor ce sunt considerate anormale din cadrul principal

Codul de stare respectă standardul HTTP1.1, mai precis *RFC7231*[?] și respectă valorile prezentate în tabelul de mai jos:

Valoarea codului	Semnificație
1xx	Apelul a fost primit, dar procesarea încă este în desfășurare
2xx	Apelul a fost primit și procesat cu succes
3xx	Clientul trebuie să execute pași suplimentari pentru ca apelul să fie procesat
4xx	Apelul este greșit din punct de vedere sintactic sau nu poate fi procesat
5xx	Apelul este valid dar serverul a întâmpinat o eroare în timpul procesării

Tabela 3.1: Descrierea codurilor returnate de către server

Modul în care este structurat serverul face foarte ușoară crearea de noi API-uri, cu aceeași funcționalitate, dar orientate spre tipuri de anomalii diferite. Așa cum a fost descris și în introducere, detectarea anomaliilor este dependentă de setul de date de referință. Astfel, fiecare model antrenat pe un set de date diferit, acoperă o arie diferită de anomalii. Altfel spus, în funcție de setul de date folosit la antrenare, sistemul va detecta un alt set de evenimente drept anomalii. În final, un nou API este necesar pentru fiecare astfel de model.

Deoarece arhitectura curentă a serverului încarcă modelele pre-antrenate dintr-o destinație S3 la momentul inițializării, pentru a crea un nou API ce a fost antrenat pe un alt set de date, este necesară doar crearea unei noi destinații S3, încărcarea modelelor pre-antrenate la acea destinație, schimbarea în cod a denumirii vechii destinații în cea nouă și apoi un nou deployment. Astfel, cu un număr minim de pași, poate fi creat un nou API ce execută etapa de inferență pentru un sistem ce a fost antrenat pe un nou set de date.

Codul ce se execută în momentul primirii pe server a unei cereri pe endpoint-ul descris mai sus este :

```
@app.route('/upload/<file_key>', methods=['POST'])
def lambda_handler(file_key):
    frame_name = file_key
    arguments_tuples = [(frame_name, s3_client)
                        ,(frame_name+"_d3", s3_client)
                        ,(frame_name+"_p3", s3_client)]

    pool = ThreadPool(processes=4)
    #descarca imaginile necesare procesarii in mod paralel
    results = pool.map(load_frame, arguments_tuples)
    frame = results[0]
    frame_d3 = results[1]
    frame_p3 = results[2]
    feature_vectors, bounding_boxes =
        get_feature_vectors_and_bounding_boxes(frame_predictor,
                                                frame,
                                                frame_d3,
```

```

frame_p3)

frame_score = 0
boxes = []

for idx, vector in enumerate(feature_vectors):
    score = frame_predictor.get_inference_score(vector)
    if score > frame_predictor.threshold:
        c1,l1,c2,l2 = bounding_boxes[idx]
        boxes.append([c1,l1,c2,l2])
    if score > frame_score:
        frame_score = score
response = {"statusCode": 200,
            "body" : frame_score,
            "boxes" : boxes}
return Response(json.dumps(response), mimetype='application/json')
```

Etapă de inferență a sistemului poate fi rulată și local, dezvoltând în acest sens un program. Acesta execută toate etapele inferenței folosind aceleași modele ca și serverul plasat în cloud, deoarece la momentul execuției programului, acesta descarcă modelele necesare din aceeași destinație S3. Astfel se asigură faptul că nu există diferențe între programul ce este executat local și cel ce deservește API-ul, iar dacă modelele sunt îmbunătățite și schimbate, această schimbare se propagă automat către ambele moduri de execuție.

3.4 Scenarii de utilizare

Acest sistem de detecție a anomaliilor are aplicații multiple în domeniul supravegherii inteligente. Este cunoscut faptul că datele înseamnă putere și control, însă acest lucru este adevărat doar atunci când datele sunt analizate. Există nenumărate sisteme de supraveghere actuale ce nu conțin nici o formă de analiză pasivă, sub forma unui program. Majoritatea se bazează pe monitorizarea activă a datelor de către o persoană, fapt ce limitează capacitatea sistemului de a fi monitorizat, deoarece pentru supravegherea unei suprafețe mari, nu este posibilă monitorizarea activă a tuturor datelor.

Deoarece sistemul actual a fost dezvoltat să ruleze folosind relativ puține resurse, acesta este pretabil pentru a fi folosit atât pe echipamente independente din industria IoT și pe sisteme de supraveghere actuale, cât și pe sisteme complexe de analiză a datelor agregate din mai multe surse.

Un scenariu de utilizare în industria IoT, ar fi detecția braconajului. Deoarece de obicei conexiunea la internet în parcurile naturale este foarte slabă, trimiterea tuturor imaginilor către server nu este o opțiune. Astfel, sistemul trebuie rulat local, pe dispozitivul propriu-zis, urmând mai apoi, în cazul detectării unei anomalii, să fie trimisă către server doar o notificare ce conține și cadrul detectat ca fiind anormal, fapt ce reduce semnificativ cantitatea de date trimisă prin rețea. În acest caz, sistemul

ar putea fi combinat cu un senzor fizic de mișcare, pentru a analiza doar imaginile în care există mișcare. Astfel, consumul de energie al dispozitivului ar fi de asemenea redus.

În ceea ce privește sistemele de analiză a datelor, sistemul actual este pretabil pentru a fi distribuit într-o rețea de servere și pentru a analiza în paralel înregistrări din surse multiple. În acest caz, sistemul poate analiza atât date în timp real, cât și date istorice în funcție de necesități. Astfel, sisteme complexe precum sistemul de supraveghere al unui oraș poate fi monitorizat pasiv, iar monitorizarea activă să se facă doar asupra anomaliilor detectate de sistem. Astfel, șanșele ca anumite evenimente să treacă neobservate sunt reduse semnificativ. Pentru sisteme de supraveghere bazate pe multe surse, o bună utilizare a sistemului ar fi folosirea acestuia pentru a construi un program de notificare automată asupra anomaliilor detectate. Astfel, imaginea în timp real a camerei pe care s-a detectat anomalia ar putea fi adusă în prim planul personalului ce realizează supravegherea activă complementară, pentru o lua măsurile ce se impun.

Capitolul 4

Execuția în cloud a sistemului

4.1 Cloud-computing în inteligență artificială

Execuția în cloud este un domeniu cu o creștere substanțială în ultimii ani, înlocuind practic o bună parte din infrastructurile clasice existente. Acest lucru are implicații și în dezvoltarea sistemelor de inteligență artificială, acesta fiind un domeniu știut drept unul cu necesar de putere de execuție mare, dar și cu un potențial de scalare extraordinar. Tocmai acest potențial, face ca inteligența artificială să fie candidatul perfect pentru execuția în cloud. Faptul că efortul de dezvoltare pentru a crea un sistem în cloud pregătit să deservească milioane de utilizatori este egal cu dezvoltarea unui sistem pentru câteva mii de utilizatori în mod clasic, arată din nou de ce această soluție a devenit alegerea perfectă pentru multe companii.

Deși opțiunile pentru execuția în cloud sunt vaste, de la sisteme complet administrate de către utilizator, până la funcții cloud complet administrate de distribuitor, tendința este ca acolo unde este posibil, clientul să se ocupe cât mai puțin de administrare, și cât mai mult de dezvoltarea propriu zisă a produsului. Un alt subiect de interes pentru execuția în cloud este comparația între arhitecturi *serverfull* și *serverless* dar alegerea între cele două diferă de la sistem la sistem deoarece pentru a alege o arhitectură *serverless*, sistemul trebuie construit pentru asta încă de la începutul dezvoltării.

Sistemele de ML, sau IA, sunt de obicei folosite pentru a îmbunătăți sisteme deja existente, sau pentru a ușura luarea deciziilor vitale. Din acest motiv, monitorizarea execuției, asigurarea disponibilității sistemului și posibilitatea de a controla costurile sunt lucruri foarte importante pentru alegerea modului de execuție a sistemului. Pe lângă asigurarea tuturor acestor facilități, soluțiile cloud oferă de multe ori și performanțe mai bune decât infrastructurile fizice echivalente, oferind astfel toate bazele necesare pentru găzduirea sistemelor IA/ML.

La ora actuală, primii 3 distribuitori de soluții cloud (Amazon, Microsoft, Google) oferă și soluții speciale de implementare rapidă a inteligenței artificiale în aplicații deja existente, folosind sisteme special gândite să suporte întreg procesul de dezvoltare și execuție, totul înglobat într-un singur serviciu :

- Amazon SageMaker
- Google AI Platform
- Azure Machine Learning

4.2 Analiza opțiunilor

Opțiunile în ceea ce privește execuția în cloud sunt diverse și în continua dezvoltare. Deși categoriile de soluții sunt bine definite, serviciile propriu zise sunt modificate destul de des, având din ce în ce mai multe facilități.

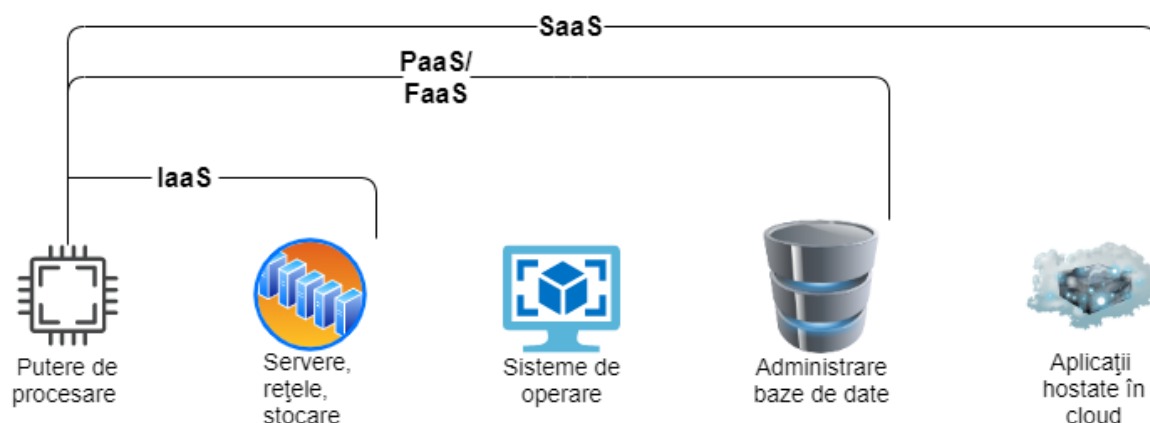


Figura 4.1: Componentele diverselor modele de procesare în cloud

Categoriile de servicii cloud sunt :

- Infrastructure as a Service (IaaS)
- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Function as a Service (FaaS)

Soluțiile de tip *IaaS* presupun punerea la dispoziția clientului doar a infrastructurii cerute, fără nici o administrare din partea distribuitorului. Din acest punct de vedere, soluțiile IaaS pot fi vazute de către client drept infrastructuri fizice, singura diferență fiind că locul în care este amplasată aparatura nu este deținută de către client. În ceea ce privește securitatea, update-urile, softurile și toate legăturile din cadrul infrastructurii, acestea sunt administrate strict de către client. Între a rula un sistem pe o infrastructură

locală și a rula un sistem pe o infrastructura aflată în cloud dar folosind o soluție IaaS există doar diferențe de natură economică.

SaaS reprezintă o metodă pentru distribuirea softurilor pe internet la cerere și pe bază de subscripție. Soluțiile SaaS sunt folosite pentru găzduirea și administrarea softurilor pentru a facilita distribuția de update-uri sau patchuri de securitate.

Soluțiile *PaaS* reprezintă un mod de a crea medii de dezvoltare, testare sau producție la cerere. Este construit pentru a asigura o modalitate rapidă de a crea aplicații software diverse, de la aplicații web, la cele mobile sau API-uri ce fac parte dintr-un alt sistem. Practic, este un mod de a crea și administra soluții IaaS în mod automat, asigurând toate uneltele necesare pentru scalabilitate automată, distribuirea traficului între servere, mentenanță, asigurarea securității și lansarea de noi versiuni. Deși este un serviciu serverfull, acesta ia de pe umerii clientului sarcina creeri și administrării infrastructurii, grăbind astfel procesul de dezvoltare și lansare a unui nou sistem software.

În ceea ce privește FaaS, acesta este un domeniu nou, deoarece a apărut pentru prima dată în 2010 fiind oferit de câteva start-upuri la acea vreme. Acest mod de dezvoltare orientat spre microservicii a devenit trendul în industrie în ultimii ani pentru sisteme cu potențial de scalare mare, deoarece prezintă numeroase avantaje din punct de vedere al modului de dezvoltare și de execuție în industrie. În momentul de față, pentru servicii de tip FaaS sunt 3 mari jucători: Amazon cu AWS Lambda, Google cu Google Cloud Functions și Microsoft cu Azure Functions.[?]. Numeroase lucrări din domeniu [?, ?] arată ca rularea algoritmilor de machine learning folosind soluții FaaS (Function as a service) precum AWS Lambda sau Google cloud functions, este în sine o problemă ce necesită soluții de optimizare a codului pentru a îndeplini restricțiile soluțiilor de rulare serverless, cum ar fi memoria limitată a mediului de execuție.

O altă caracteristică a soluțiilor cloud este tipul de execuție și anume: *serverfull* sau *serverless*. Prin serverfull, se înțelege o soluție ce rulează pe servere ce pot fi indentificate în mod unic, care rulează în mod continuu, dar pe care se execută diferite operații în funcție de nevoile sistemului. IaaS și PaaS sunt mereu soluții serverfull, în timp ce un serviciu SaaS poate fi atât serverfull cât și serverless. Prin serverless se înțelege o soluție ce lansează noi micro-instanțe pentru fiecare cerere, ce este mai apoi oprită după execuția codului. Costul unei soluții serverless este direct proporțional cu timpul de execuție, în timp ce pentru soluțiile serverfull costul este constant.

Caracteristicile celor 2 tipuri sunt prezentate în tabelul de mai jos :

Caracteristici	Soluții AWS serverless	Soluții AWS serverfull
Când este activ serviciul	Când este declanșat de un eveniment	Continuu, până la oprire
Limbaaj de programare	Python, Java, Go, C#, și altele...	Orice limbaj
Max RAM	0.125 - 3 Gb	0.5-1952 Gb
Max Capacitate de stocare	0.5 Gb	0-3600 Gb
Max Timp de rulare	900 secunde	Nelimitat
Unitatea minimă taxată	0.1 secunde	60 de secunde
Preț minim pe unitate	\$0.0000002	\$0.0000867
Sistem de operare	Ales de distribuitorul de soluții cloud	Ales de către client

4.3 Comparație între soluțiile PaaS si FaaS

Atât soluțiile PaaS cât și cele FaaS sunt pretabile sistemelor de inteligență artificială, deoarece oferă administrare automată, putere mare de calcul și avantajul că suportă optimizari prin rularea pe GPU.

Soluțiile PaaS, deși oferă servicii de administrare automată, în spatele acestui mecanism se ascunde tot o infrastructură clasică, cu servere fizice ce rulează în mod continuu, ce necesită echilibrarea traficului între acestea, lansarea de noi servere, lansarea de noi versiuni a aplicației pe serverele deja pornite sau închiderea unor servere atunci când nu mai sunt folosite. Toate acestea sunt realizate de către distribuitorul de soluții cloud, punând la dispoziția clientului un mediu gata să suporte orice tip de trafic către aplicație. Astfel dezvoltatorul se poate gândi doar la construirea unui server cât mai eficient, pentru a folosi resursele unei instanțe la maxim. Un aspect important pentru soluțiile PaaS este costul asociat instanțelor folosite. Deși numărul lor este variabil, în orice moment măcar o instanță trebuie să fie activă. Asta înseamnă că dacă sistemul poate rula doar pe instanțe cu resurse suplimentare, ce au un cost orar mai mare, sistemul va avea un cost de operare mare chiar și atunci când este subutilizat. Un sistem ce folosește soluții PaaS în mod ideal este de preferat să ruleze pe mai multe instanțe cu capacitate de execuție mică față de mai puține instanțe cu capacitate mare, deoarece lansarea sau închiderea de noi instanțe este gratuită, în timp ce rularea acestora atunci când sunt sub-utilizate nu este.

Soluțiile FaaS, pe de altă parte, abstractizează și mai mult infrastructura existentă, luând orice responsabilitate de pe umerii clienților atunci când vine vorba de administrarea acestora. FaaS presupune existența unei funcții, ce simbolizează o aplicație, ce este executată de fiecare dată când apare un eveniment declanșator. Avantajul acestei abordări este costul asociat, și anume faptul că plata se realizează strict pentru timpul în care funcția a fost executată. Timpul petrecut pentru a aștepta apariția unui eveniment nu este taxat. Spre exemplu, soluția FaaS a celor de la Amazon, AWS

Lambda, poate fi declanșată în numeroase moduri, de la încărcarea unui fișier într-o destinație S3 prestabilită, la un apel HTTP către un endpoint atașat funcției. Aceste soluții suportă majoritatea limbajelor de programare, și pot fi configurate în așa fel încât să acopere orice nevoie în ceea ce privește puterea de execuție. Soluțiile FaaS prezintă și o serie de dezavantaje printre care se numără:

- Există o limită asupra dimensiunii codului ce urmează să fie executat de către funcție.
- Timpul de execuție a unei funcții este de asemenea limitat.
- Starea programului nu este păstrată între execuții făcând necesară stocarea datelor ce trebuie să persiste în locații dedicate (S3 sau o bază de date)
- Limitarea executării concurente. (Lambda nu permite mai mult de 1000 de funcții să fie executate concomitent)

Deși din descrierea acestui tip de soluții, pare alegerea ideală pentru execuția etapei de inferență pentru un model de *machine learning*, situația devine complicată atunci când sistemul este unul complex, iar respectarea acestei limite de dimensiune devine un blocaj. De multe ori, aplicațiile de AI/ML depind de biblioteci și frameworkuri cunoscute în domeniu precum *tensorflow*, *keras*, *mxnet*, dar care au fost dezvoltate având în minte eficiența execuției, și nu minimizarea codului sursă. Având în vedere că AWS Lambda limitează dimensiunea codului atașat la 250MB, lansarea sistemelor ce se bazează pe rețele neuronale este o sarcină grea din punct de vedere al încadrării în aceste restricții.

4.4 Descrierea soluției curente

Soluția aleasă pentru execuția etapei de inferență în cloud este *AWS Elastic Beanstalk*. Acest serviciu de tip PaaS oferă mediul perfect pentru lansarea aplicației, atât din punct de vedere al configurabilității cât și al securității oferite. Pentru a lansa o aplicație folosind elastic beanstalk, este necesar să creezi un pachet de lansare, ce conține codul sursă și celelalte instrucțiuni pentru inițializarea serviciului. Deși acesta este limitat la 512MB, este mai mult decât îndeajuns deoarece mediul de rulare și dependențele programului nu sunt adăugate în acest pachet, ci sunt descărcate pe server la momentul lansării pe baza unui fișier de configurare adăugat în pachet numit *requirements.txt*.

În timpul lansării aplicației, sunt create automat următoarele resurse :

- O flotă , sau o singură instanță *EC2*, în funcție de configurație
- Un *Elastic Load Balancer* ce distribuie traficul între instanțe

- Un *AutoScaling Group* ce definește regulile de scalare a numărului de instanțe
- O destinație *S3* în care este stocat codul sursă al aplicației
- Și un *API Gateway* ce joacă rolul de punct de intrare al traficului în aplicație

Deoarece la bază se află instanțe EC2, acestea oferă un mare avantaj din punct de vedere al execuției concurente acestei soluții față de AWS Lambda sau alte soluții FaaS asemănătoare. Deși mai multe instanțe Lambda pot fi executate concurrent, în cadrul unei singure instanțe, apelurile concurente nu sunt permise. Luând în calcul că în etapa de inferență sunt și părți de input/output în care resursele sunt nefolosite, folosirea instanțelor EC2 ce pot rula mai multe apeluri în mod concurrent este opțiunea optimă din punct de vedere al folosirii resurselor.

Capacitatea de multi-threading din cadrul serverelor, la care se adaugă scalarea automată a numărului acestora în funcție de traficul ce accesează API-ul, face ca resursele să fie folosite la maxim, atât din punct de vedere al performanței disponibile, cât și din punct de vedere economic.

Un alt avantaj al acestei soluții este ușurința cu care se poate schimba mediul de execuție. Spre exemplu, într-un anumit scenariu, în care se observă în panoul de monitorizare că instanțele EC2 curente sunt suprasolicitate, cu ajutorul interfeței web, sau celei din linia de comandă, pot fi schimbate tipurile de instanțe EC2 folosite cu unele mai performante fără a avea serviciul indisponibil pe durata acestei modificări.

Capitolul 5

Rezultate și Concluzii

Sistemul a fost evaluat pe setul de date *Avenue*. Ca și modalitate de evaluare a fost luată în considerare aria de sub curbă (eng: *area under the curve (AUC)*) raportată la scorul fiecărui cadru.

Scorul unui cadru reprezintă maximum dintre scorurile obținute de obiectele din imagine. Această valoare este valoroasă și în practică, deoarece pe baza acesteia se pot izola cadrele ce au un scor de anomalie peste un anumit nivel. Pentru a evalua întregul set de date, au fost evaluate individual video-urile de test, iar mai apoi a fost calculată media scorurilor obținute, metodă folosită și de către *Ionescu et al.* [?].

Rezultatele sistemului sunt de (65 %), sub cele prezentate în lucrarea inițială [?], datorate orientării spre viteză pe sisteme ce nu posedă GPU, limitărilor fizice ale mașinii folosite pentru antrenare și a folosirii unei librării diferite pentru etapa de inferență.

Așa cum se observă și în figura 5.1 sistemul izolează cu succes evenimentele anormale. Totuși, detecția obiectelor rămâne cel mai mare dezavantaj al sistemului, deoarece, atât din punct de vedere al timpului de execuție, cât și al performanței, acesta este locul în care sistemul poate fi îmbunătățit. Față de sistemul de referință [?], alegerea unui nou tip de detector de obiecte optimizat pentru execuția pe CPU, a crescut viteza de la 0.8 FPS la 8 FPS pe sisteme ce nu posedă GPU, ceea ce reprezintă o îmbunătățire semnificativă a sistemului. Însă, deși mai rapid, noul detector nu realizează operația de segmentare la fel de bine pentru obiectele mici, sau îndepărtate de cameră lucru ce

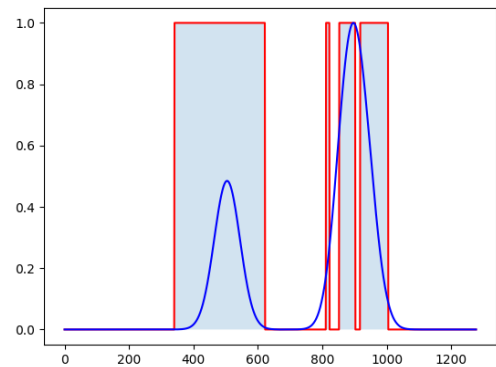


Figura 5.1: Scorurile la nivel de cadru pentru video-ul nr.6 din setul de date *Avenue*. Scorul fiecărui cadru este marcat cu albastru închis, în timp ce zonele cu fundal albastru deschis reprezintă perioadele în care au existat anomalii.

face ca pe setul de date Avenue, să nu obțină performanțe la fel de bune ca sistemul inițial.

Majoritatea detecțiilor fals pozitive ale sistemului sunt în mare parte cauzate de segmentarea imperfectă a detectorului de obiecte(fig. 5.2). În urma experimentelor a reieșit ca situațiile în care mai multe se află în același cadran, sau 1 obiect nu este încadrat complet de către cadran pun probleme sistemului și generează detecții fals pozitive. În următoarele versiuni ale sistemului se vor încerca modalități de filtrare și validare a detecțiilor de obiecte, pentru a elimina aceste probleme.



Figura 5.2: Detecții fals pozitive datorate detecției deficitare a marginilor obiectelor de către detectorul de obiecte. În prima imagine se poate observa un cadran ce conține 2 persoane, în timp ce în cea de a 2-a imagine, cadranul ce ar fi trebuit să încadreze persoana în totalitate, conține doar o mică parte din aceasta.

Pentru a demonstra capacitățile sistemului de a detecta anomalii în absența erorilor de detecție a obiectelor, am înregistrat un nou set de date, ce conține persoane în plan apropiat, simulând o sală de așteptare de capacitate redusă (e.g. cabinet dentar, secretariat, clinica de dimensiune redusă, etc.). Acest set de date conține 4 videoclipuri de antrenament ce reprezintă comportamentul normal al subiecților, și 4 videoclipuri de test în care sunt exemplificate diverse comportamente anormale, precum :

- Alergare
- Confruntări
- Leșin
- Vandalism

Rezultatele empirice obținute pe acest set de date, sunt satisfăcătoare, fiind detectate toate evenimentele anormale.

5.1 Timpi de execuție și necesarul de resurse

Din punct de vedere al timpului de execuție al întregului sistem, acesta necesită 130 de milisecunde, din care 120 de milisecunde sunt petrecute detectând obiecte, și

10 milisecunde trecând obiectele prin encoder și apoi prin clasificatorul final. Astfel, sistemul evaluează cadrele cu o viteză de 8 cadre pe secundă. În ceea ce privește accesarea API-ului, sunt necesare în medie 1.1 secunde pentru încărcarea imaginilor ce vor fi procesate în cloud, și 0.7 secunde pentru execuția apelului HTTP propriu zis.

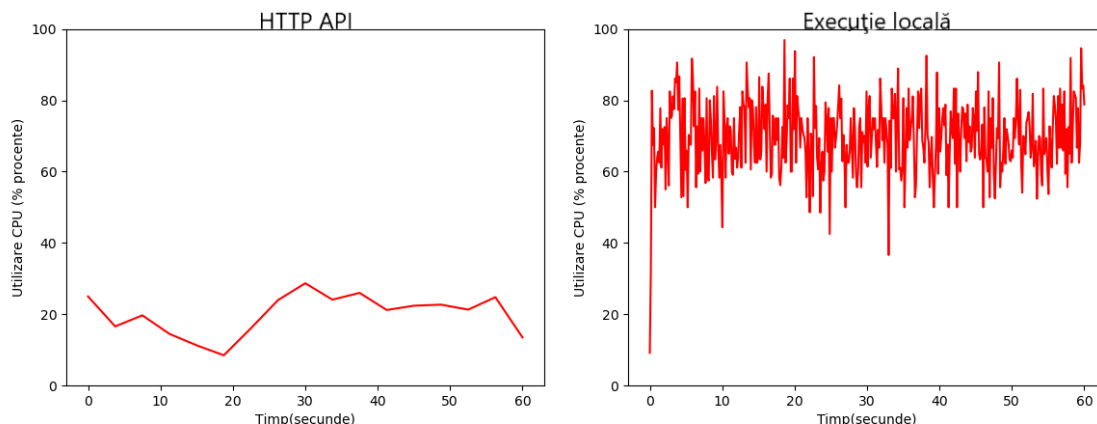


Figura 5.3: Comparație între resursele folosite în timpul execuției sistemului timp de 1 minut, execuție locală versus folosind API-ul.

Din punct de vedere al necesarului de resurse, am comparat cele 2 moduri de utilizare a sistemului : prin rulare locală, sau prin accesarea HTTP API-ului. Rezultatele prezentate în figura 5.3 demonstrează economia de resurse făcută accesând API-ul. Astfel, sistemul poate fi folosit în 2 moduri :

- Rapid dar ce folosește intens procesorul sistemului. În acest mod se pot rula și analize în timp real.
- Lent dar cu un necesar foarte scăzut de resurse. Acest mod poate fi folosit atunci când viteza de analiză nu este importantă.

Menționez ca toate măsurătorile au fost realizate pe un sistem cu i7-6600U 2.6 GHz CPU și 16 GB RAM.

5.2 Concluzii

Sistemul de detectare a anomaliilor dezvoltat a îndeplinit așteptările inițiale, îmbunătățind viteza pe sisteme ce posedă resurse limitate. De asemenea, a fost creat un cadru ce face posibilă convertirea sistemelor clasice de supraveghere în sisteme inteligente, prin rularea sistemului de detecție a anomaliilor fie local, fie prin intermediul arhitecturii client-server.

În versiunile viitoare, se propune îmbunătățirea preciziei sistemului, menținând viteza curentă de execuție.

Listă de figuri

1.1	Statistică ce evidențiază importanța domeniului cloud computing în ultimii ani	4
2.1	Aplicarea operatorului de convoluție	8
2.2	Arhitectura abstractizată a API-ului	12
2.3	Pagina de start a aplicației	13
3.1	Prezentarea principiului de funcționare a autoencoderelor și arhitectura lor generală	17
3.2	Arhitectura primei etape de antrenare	20
3.3	Gradienti obținuți în urma aplicării unui kernel 3×3 , respectiv unul de dimensiune 21×21	21
3.4	Arhitectura etapei finale de antrenare	22
3.5	Arhitectura etapei de inferență	23
4.1	Componentele diverselor modele de procesare în cloud	29
5.1	Scorurile la nivel de cadru pentru video-ul nr.6 din setul de date Avenue. Scorul fiecărui cadru este marcat cu albastru închis, în timp ce zonele cu fundal albastru deschis reprezintă perioadele în care au existat anomalii.	34
5.2	Detecții fals pozitive datorate detecției deficitare a marginilor obiectelor de către detectorul de obiecte. În prima imagine se poate observa un cadran ce conține 2 persoane, în timp ce în cea de a 2-a imagine, cadranul ce ar fi trebuit să încadreze persoana în totalitate, conține doar o mică parte din aceasta.	35
5.3	Comparație între resursele folosite în timpul execuției sistemului timp de 1 minut, execuție locală versus folosind API-ul.	36

Listă de tabele

3.1	Descrierea codurilor returnate de către server	25
-----	--	----