

UNIVERSITATEA “ALEXANDRU IOAN CUZA” IAȘI

**FACULTATEA DE INFORMATICĂ**



LUCRARE DE LICENȚĂ

**Învățare nesupervizată din video-uri**

propusă de

**Cotofană Victor**

**Sesiunea:** iulie, 2019

Coordonator științific

**Lect.dr. Ignat Anca**

UNIVERSITATEA “ALEXANDRU IOAN CUZA” IAȘI

**FACULTATEA DE INFORMATICĂ**

**Învățare nesupervizată din video-uri**

**Coțofană Victor**

**Sesiunea:** iulie, 2019

Coordonator științific

**Lect.dr. Ignat Anca**

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele \_\_\_\_\_

Data \_\_\_\_\_ Semnătura \_\_\_\_\_

### **DECLARAȚIE privind originalitatea conținutului lucrării de licență**

Subsemnatul(a) .....

domiciliul în .....

născut(ă) la data de ....., identificat prin CNP .....,

absolvent(a) al(a) Universității „Alexandru Ioan Cuza” din Iași, Facultatea de .....

specializarea ....., promoția ....., declar pe

propria răspundere, cunoscând consecințele falsului în declarații în sensul art. 326 din Noul

Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art.143 al. 4 și 5 referitoare la

plagiat, că lucrarea de licență cu titlul:

\_\_\_\_\_

\_\_\_\_\_elaborată sub îndrumarea dl. / d-na

\_\_\_\_\_, pe care urmează să o susțină în fața comisiei

este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diploma sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data azi, .....

Semnătură student .....

## DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Învățare nesupervizată din video-uri*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași,

Absolvent *Coțofană Victor*

---

(semnătura în original)

## Cuprins

<b>Introducere .....</b>	<b>6</b>
<b>Contribuții .....</b>	<b>7</b>
<b>Capitolul 1. Învățare nesupervizată din video-uri .....</b>	<b>8</b>
<b>Capitolul 2. Arhitectură și implementare .....</b>	<b>10</b>
2.1 Rețele neuronale .....	10
2.2 Rețele neuronale convolutive .....	10
2.3 Rețelele siameze .....	12
2.4 Arhitectura detaliată a modelului .....	12
2.5 Implementarea în Keras.....	15
<b>Capitolul 3. Preprocesarea video-urilor.....</b>	<b>19</b>
3.1 Mulțimea de date folosită .....	19
3.2 Algoritmul de preprocesare .....	20
<b>Capitolul 4. Mașina virtuală și antrenarea modelului.....</b>	<b>23</b>
4.1 Google Cloud Platform .....	23
4.2 API-uri Google .....	24
4.3 Setarea mașinii virtuale .....	25
4.4 Alegerea numărului de categorii și numărului de iterații .....	27
4.5 Antrenarea modelului și problemele eventuale .....	28
<b>Capitolul 5. Evaluarea modelului și vizualizarea rezultatului.....</b>	<b>29</b>
<b>Concluzii .....</b>	<b>35</b>
<b>Bibliografie .....</b>	<b>36</b>

## Introducere

În ultimii ani domeniul inteligenței artificiale și în special învățarea automată au avut o creștere semnificativă pe plan mondial. O mare parte a atenției industriei și a domeniului este axată pe partea de Deep Learning a domeniului, aceasta aducând rezultate excepționale. Cu ajutorul acesteia, s-au creat modele care “învăță” mult mai rapid și mai bine în comparație cu un creier uman, oferind rezultate mult mai bune chiar și față de specialiștii dintr-un anumit domeniu.

Fiind într-o era a informației, în care fluxul de intrare și creare a conținutului depășește capacitatea de procesare a acesteia, apare dorința și necesitatea de a găsi soluții mult mai eficiente decât cele actuale. Un exemplu de o astfel de problemă este fluxul de încărcare de video-uri pe platforma YouTube. Conform statisticilor în fiecare minut pe platformă sunt încărcate în jur de 500 de ore de conținut video nou<sup>1</sup>. Problema nu constă în stocarea acestor video-uri, dar cum determini ce conținut apare în aceste video-uri în cel mai scurt timp posibil? Problema nu se pune doar în scopul creării categoriilor pentru utilizatori, dar și pentru filtrarea video-urilor interzise pe platformă. Având problema de față și având toate progresele în domeniul inteligenței artificiale și de Deep Learning, poate fi propusă o posibilă soluție a acestei probleme.

Lucrarea de față prezintă implementarea unui model de rețea neuronală care este antrenată să clasifice video-uri nefolosind etichetele (eng. *label*) acestora de antrenare, fiind astfel o învățare complet nesupervizată. Modelul prezentat este bazat pe metoda propusă de Redondo-Cabrera et. al. 2018 în articolul *Unsupervised learning from videos using temporal coherency deep networks*.<sup>2</sup>

În **Capitolul 1** este descrisă problema învățării nesupervizate din video-uri și ce diferă față de rețelele neuronale tradiționale și învățarea supervizată.

În **Capitolul 2** este prezentată în detaliu arhitectura și implementarea modelului. De asemenea sunt introduse conceptele teoretice folosite în arhitectură.

În **Capitolul 3** este descrisă mulțimea de date folosită pentru antrenarea modelului și algoritmul necesar de preprocesare a datelor pentru model.

În **Capitolul 4** sunt prezentate procesul de setare a mașinii virtuale cât și API-urile terțe folosite.

În **Capitolul 5** sunt prezentate rezultatele, argumentarea și vizualizarea acestora și comparația cu rezultatul din articolul original.

---

<sup>1</sup> <https://expandedramblings.com/index.php/youtube-statistics/>

<sup>2</sup> <https://www.sciencedirect.com/science/article/pii/S1077314218301772>

## Contribuții

Pentru realizarea lucrării aveam nevoie de un anumit fundal de cunoștințe pentru a înțelege articolul mai bine. Fiind un domeniu nou pentru mine și având doar cunoștințele de bază privitoare la rețelele neuronale, am avut nevoie de mai multe informații și resurse. Pentru acestea, am urmărit primele 2 săptămâni ale cursului lui Andrew Ng a tutorialului online *Convolutional Neural Networks*<sup>3</sup> de pe platforma online de învățare Coursera.

Având cunoștințele teoretice necesare, următorul pas a fost să acumulez cunoștințe de implementare a acestora. Pentru aceasta am folosit Keras, o bibliotecă open-source pentru rețele neuronale scrisă în Python<sup>4</sup>. O introducere bună pentru această bibliotecă sunt tutorialele de pe site-ul oficial Tensorflow<sup>5</sup>.

Pentru antrenarea rețelei neuronale am optat pentru o mașină virtuală cu placă video. Aceasta a sporit considerabil viteza cu care modelul învață. Pentru a configura mașina virtuală să fie capabilă să ruleze framework-ul pe placa video a fost necesară instalarea anumitor drivere specifice programării pe placa video.

Pentru determinarea hyperparametrilor, a fost necesară consultarea codului sursă a autorilor articolului original<sup>6</sup>. Pentru a monitoriza procesul învățării am afișat la fiecare iterație într-un spreadsheet online detaliile relevante. Neavând control deplin asupra mașinii virtuale, și având în vedere că procesul de învățare ia mult timp, am optat pentru salvarea modelului la anumite perioade.

În final după învățarea modelului și având rezultatele, am creat o reprezentare a datelor care ușurează evaluarea modelului.

---

<sup>3</sup> <https://www.coursera.org/learn/convolutional-neural-networks>

<sup>4</sup> <https://keras.io/>

<sup>5</sup> <https://www.tensorflow.org/beta/guide/keras/overview>

<sup>6</sup> <https://github.com/gramuuh/unsupervised>

## Capitolul 1. Învățare nesupervizată din video-uri

Majoritatea modelelor de rețele neuronale au un proces de învățare supervizat. Mulțimea de date cu care este antrenat modelul conține etichete (eng. *labels*) care indică ceea ce reprezintă acele date, elementul esențial pe baza căruia modelul învață. În asta și constă învățarea supervizată, când datele de intrare sunt cunoscute și ceea ce reprezintă ele (eng. *ground truth*) este cunoscut și ajută la învățare.

Învățarea nesupervizată, pe de altă parte, nu folosește etichete pentru învățare, și atât arhitectura cât și modul de învățare diferă. Astfel un model poate învăța o anumită mulțime de date fără să știe efectiv ce reprezintă acea mulțime de date. Ceea ce reprezintă un avantaj enorm față de învățarea supervizată, care necesită o mulțime de date etichetată. Procesul de etichetare ia foarte mult timp pentru că e făcut manual. Astfel creând un model ce învață fără etichete, eliminăm măcar pentru o mare parte, procesul de etichetare inițială a mulțimi de date.

Pentru a crea o rețea neuronală care învață nesupervizat din video-uri, am unit două arhitecturi de rețele în una singură. Și anume, am creat o rețea siameză ce conține două rețele neuronale convolutive (eng. *Convolutional Neural Network*). Rețelele convolutive conțin o arhitectură ce facilitează învățarea supervizată a imaginilor, iar rețelele siameze au ca idee de bază învățarea nesupervizată pe baza diferenței reprezentărilor a două arhitecturi de rețele neuronale.

Pentru a le combina am eliminat ultimul nivel din arhitectura rețelei convolutive, care deseori e un SoftMax, și am oferit către funcția de învățare din rețeaua siameză, care e *contrastive loss function*, rezultatul penultimului nivel, care reprezintă lista cu trăsături a datelor de intrare. Având două rețele convolutive avem 2 reprezentări ale datelor care sunt introduse în funcția contrastive loss care învață cât de asemănătoare sau nu sunt acestea, astfel învățând nesupervizat mulțimea de date de intrare.

Mulțimea de date de intrare este constituită din video-uri. Antrenarea modelului pe baza unor video-uri este realizată în următorul mod. Din fiecare video sunt extrase cadrele necesare procesării, și apoi sunt introduse în model fie ca perechi pozitive, cadre din același video, fie ca perechi negative, cadre din video-uri diferite. Din 5 perechi de cadre procesate, unul va fi negativ, astfel învățarea modelului se va axa mai mult pe apropierea trăsăturilor în spațiul euclidian a cadrelor din același video, decât pe depărtarea cadrelor din video-uri diferite. Dacă am fi ales exact opusul, adică 4 perechi negative și una pozitivă, modelul nu ar învăța mai nimic, rezultatul fiind foarte împrăștiat în spațiul euclidian.



Scopul e să antrenăm modelul să plaseze cadrele din același video aproape în spațiul de reprezentare a trăsăturilor și să plaseze la o anumită distanță cadrele din video-uri diferite.

Astfel modelul învață să recunoască acțiunea prezentată în fiecare cadru a video-ului și să o plaseze în spațiul euclidian al reprezentării trăsăturilor cât mai aproape de acțiunile care sunt din aceeași categorie cu video-ul inițial.

Pe scurt experimentul de față constituie din următoarele. Fiind necesară puterea computațională pentru a putea oferi rețelei timpul necesar învățării am decurs la soluții cloud. Inițial am descărcat mulțimea de date și am preprocesat-o. Mulțimea de date conține etichetele pentru învățarea supervizată, însă nu au fost folosite pentru învățare, ci doar pentru evaluare. Apoi am pornit învățarea modelului. Modelul conține un generator care parcurge o listă de video-uri care trebuie învățate, și creează perechile de cadre necesare învățării. După ce modelul a învățat toate video-urile, am rulat un alt generator care parcurgea o listă de video-uri de testare și extrăgea reprezentarea trăsăturilor din cadrele respective. Am evaluat modelul pe baza listei de reprezentări de trăsături, pe baza căreia am aflat entropia condiționată, care reprezenta performanța de descoperire a modelului, și am creat o reprezentare vizuală a rezultatelor.

## Capitolul 2. Arhitectură și implementare

### 2.1 Rețele neuronale

O rețea neuronală artificială (eng. *Artificial Neural Network*), așa cum sugerează și numele, este o versiune simplificată a modelului de rețea neuronală care există în creierul uman. Fiind inspirată dintr-un model complex, rețelele neuronale prezintă un instrument foarte puternic în domeniul Învățării automate (eng. *Machine Learning*).

Elementul de bază care prezintă blocul de construcție a rețelelor neuronale este neuronul. Neuronul poate fi vizualizat ca o funcție cu mai multe input-uri care returnează un output în dependență de importanța fiecarui input. Această „importanță” o reprezintă ponderile (eng. *weights*) acestui neuron.

Mai mulți neuroni conectați între ei, adică output-ul unuia reprezintă input-ul altuia, formează nivelele (eng. *layers*) rețelelor neuronale. Neuronii din același nivel nu sunt conectați între ei. Rețelele conțin mai multe tipuri de nivele: nivelul de input, nivelele ascunse și nivelul de output. În nivelul de input sunt introduse datele de intrare, iar nivelul de output returnează rezultatul rețelei. Nivelele ascunse mențin și învață trăsături pe care nu le știm despre input și care ajută la aflarea output-ului.

Rețeaua, folosind algoritmul *Backpropagation* ajustează ponderile neuronilor, pe baza prezicerii inițiale, pentru a îmbunătăți reprezentarea de trăsături ale inputurilor astfel învățând mai bine să returneze output-ul dorit pentru input-uri.

### 2.2 Rețele neuronale convolutive

Rețelele neuronale prezintă o soluție eficientă pentru multe probleme însă nu sunt potrivite pentru învățarea imaginilor. Rețelele fiind interconectate între ele, iar imaginile sunt de regula de dimensiuni mari, numărul de parametri pe care rețeaua ar trebui să îi învețe ar crește exponențial cu fiecare nivel nou adăugat. Astfel au fost create rețelele convolutive.

O rețea neuronală convolutivă este o rețea care conține unul sau mai multe nivele convolutive. O convoluție reprezintă o operație asupra a două funcții care produce a treia funcție care exprimă cum forma uneia a fost modificată de forma cealalte funcții. Termenul convoluție (eng. *convolution*) reprezintă atât rezultatul funcției cât și procesul calculării acesteia.<sup>7</sup>

Într-o rețea neuronală operația de convoluție este aplicată input-ului pentru a micșora dimensiunile acestuia. În cazul rețelelor convolutive, operația de convoluție transformă reprezentarea

---

<sup>7</sup> <https://en.wikipedia.org/wiki/Convolution>

unei imagini dintr-o matrice ce conține mai multe canale ale imaginii, într-o matrice mai mică. Acest proces este obținut cu ajutorul unui filtru, reprezentând o matrice mai mică. Se parcurge matricea inițială cu un anumit pas (eng. *stride*) și se crează o matrice nouă formată din produsul scalar al filtrului cu fiecare porțiune de matrice inițială de dimensiunea acestuia.

Convoluția precum am spus micșorează dimensiunile matricei. Dacă acest proces este repetat de mai multe ori, la fiecare nivel, se poate ajunge la dimensiuni foarte mici din care nu se poate învăța. Pentru a soluționa această problemă există noțiunea de *padding*. Aceasta reprezintă procesul de adăugare a pixelilor, de obicei cu valoare 0, în jurul imaginii originale pentru a menține dimensiunile sale, sau pentru a nu micșora prea tare dimensiunile matricei după convoluție.

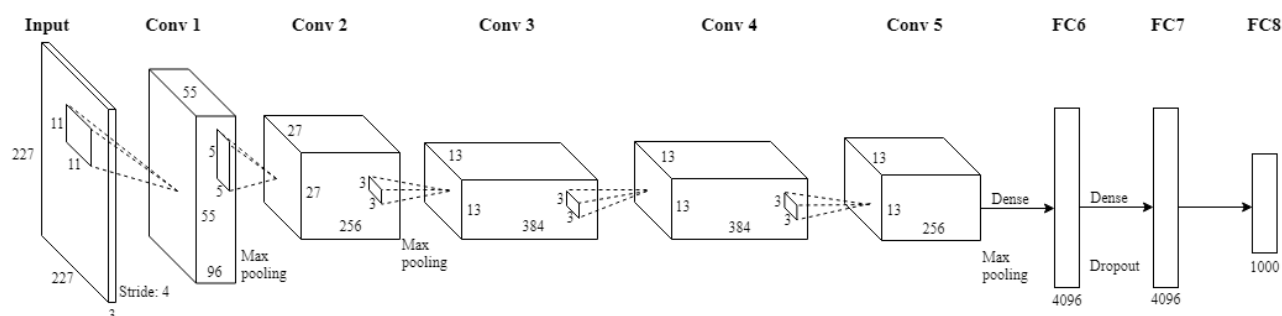
Având nivelele de convoluție, rețeaua necesită în continuare micșorarea spațiului de reprezentare astfel micșorând numărul de parametri și numărul de operații necesare rețelei. Acesta este scopul nivelelor pooling (eng. *pooling layers*). Pe lângă rezolvarea problemei menționate anterior, acestea scot în evidență anumite trăsături. Sunt două tipuri de pooling: *Max Pooling* și *Average Pooling*. Ambele funcționează pe același principiu ca filtrul la convoluție, însă în loc să facă produsul scalar fac o operație diferită. *Max Pooling* ia cea mai mare valoare ce o acoperă filtrul și o inserează în matricea nouă, iar *Average Pooling* face o medie a tuturor valorilor acoperite de filtru și o setează în matricea nouă.

O arhitectură a rețelei convolutive constă din mai multe nivele convolutive ce pot conține nivele de pooling, iar la final există nivele conectate complet (eng. *fully connected*) printre care și nivelul de output. De obicei pe nivelul de output, care reprezintă ultimul nivel, se aplică funcția *SoftMax*. Aceasta normalizează output-ul într-un vector de valori ce respectă o distribuție care asigură că suma elementelor este 1.

Una din arhitecturile clasice ale rețelelor neuronale convolutive este AlexNet<sup>8</sup>. Aceasta a crescut în popularitate după ce a câștigat concursul ImageNet 2012 (ILSVRC2012) cu rata de eroare de 15.3%, care reprezenta o îmbunătățire considerabilă față de rata de eroare de 26.2% a competitorului de pe locul 2. Arhitectura acestuia poate fi observată în **Figura 1**. Funcția de activare folosită în arhitectură este ReLU (*Rectified linear unit*). Matematic este definită ca și  $y = \max(0, x)$ .

---

<sup>8</sup> <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>



**Figura 1.** Arhitectura AlexNet

## 2.3 Rețelele siameze

Rețelele neuronale convolutive sunt foarte folositoare pentru învățarea și detectarea de imagini. Dar pentru învățare au nevoie de o mulțime mare de date etichetate, care nu mereu ușor de realizat, și nu poate fi ușor extinsă, asta presupunând reînvățarea întregului model. Clasificarea one-shot oferă soluții pentru aceste două probleme.

Clasificarea one-shot este o rețea care nu clasifică imaginile direct în clase predefinite, dar învață similaritatea dintre acestea. Pentru a face asta sunt necesare doua imagini pentru intrare în loc de una, precum sunt la rețelele convolutive. Astfel apare noțiunea de rețea siameză.

Rețelele siameze sunt constituite din două rețele convolutive care împărtășesc ponderile, astfel fiind aceeași rețea folosită de două ori, care sunt unite la un anumit nivel. În ambele rețele convolutive nu există nivelul *SoftMax*, deseori găsit în acest tip de arhitecturi, dar ultimul nivel reprezintă o listă cu trăsăturile din acea rețea. Nu este nevoie de *SoftMax* pentru că nu se clasifică datele de intrare ci este nevoie de encodarea acestora, adică lista cu trăsături, pe baza carora rețeaua siameza învață. Funcția de învățare, denumită *contrastive loss function*, determină cât sunt de apropiate sau de depărtate trăsăturile celor două date de intrare. De obicei această distanță este distanța euclidiană între cele două liste, dar poate fi folosită și oricare altă funcție. Folosind această funcție, modelul învață să returneze trăsături apropiate în spațiul euclidian pentru datele de intrare asemănătoare, de exemplu două cadre din același video, și respectiv depărtate pentru datele de intrare diferite, cadrele din video-uri diferite.

Nefolosind etichetele datelor de intrarea rețelele siameze sunt potrivite pentru învățarea nesupervizată.

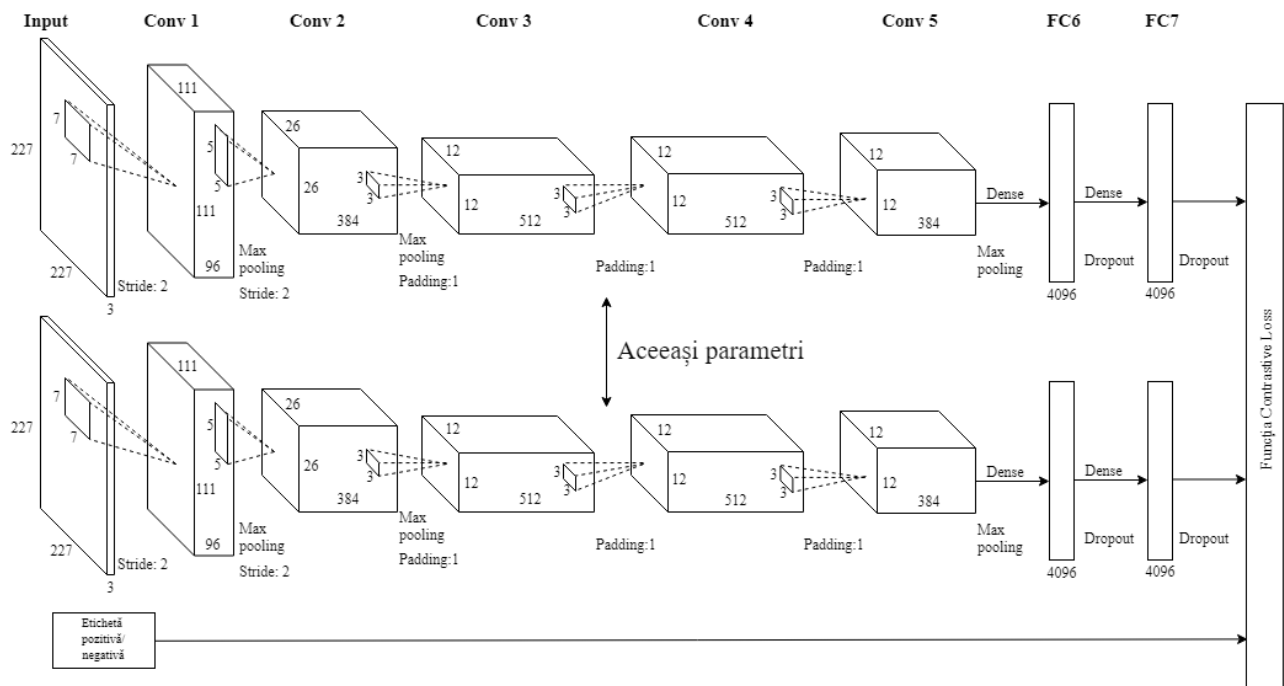
## 2.4 Arhitectura detaliată a modelului

Acum că am introdus, fiecare în parte, noțiunile de rețea neuronală convolutivă, arhitectura AlexNet și arhitectura de rețea neuronală siameză, putem prezenta arhitectura folosită în modelul de

antrenare. Modelul constă din două rețele neuronale convolutive, cu arhitectura inspirată din AlexNet, unite într-o rețea siameză. Parametrii celor două rețele sunt aceiași, astfel putem antrena modelul pe o singură rețea, în același timp salvând numărul de operații.

Inputul modelului este format dintr-o pereche de cadre, și o etichetă care reprezintă relația dintre aceste două cadre, fie pozitivă fie negativă. Pozitivă înseamnă că perechea de cadre este din același video și cadrele sunt succesive, adică unul după celălalt, iar negativă înseamnă că perechea conține cadre ce fac parte din video-uri complet diferite.

Arhitectura rețelei neuronale convolutive folosită este bazată pe arhitectura AlexNet, având aceeași dimensiune pentru intrare și nivele conectate (eng. *fully connected*) la sfârșit, dar nivelele intermediare sunt diferite. Aceasta poate fi observată în **Figura 2**. Arhitectura este inspirată din codul sursă al articolului original. Cel mai probabil autorii au decis să folosească o astfel de arhitectură pentru a mări capacitatea rețelei de a extrage trăsături cât mai diferite.



**Figura 2.** Arhitectura modelului de antrenare

Toate nivelele au același tip de inițializare a ponderilor, și anume generarea aleatoare cu distribuția normală cu deviația standard de 0.01. Având ponderile inițializate aleator de la început, modelul învață de la zero, astfel având o antrenare pur nesupervizată.

*Bias*-urile folosite sunt peste tot constante cu valoarea de 0.1, ceea ce e diferit față de arhitectura AlexNet originală, care folosea fie 0 fie 1 în dependență de nivel.

Numărul de iterații folosite pentru antrenarea modelului este discutat în **Capitolul 4. Mașina virtuală și antrenarea modelului.**

Funcția de activare este aceeași ca și în arhitectura AlexNet, adică funcția ReLu. Modelul învață folosind funcția de optimizare *Stochastic gradient descent* (SGD). Rata de învățare inițială (eng. *base learning rate*) este de 0.001 și *decay*-ul (cât de mult scade rata de învățare cu fiecare iterație) este 0.1. O adițiune a algoritmului SGD este *Momentum*, care accelerează învățarea modelului în direcția potrivită, ajungând mai rapid la convergență. Pentru antrenare, valoarea momentum-ului este de 0.9.

Rețelele convolutive se unesc la ultimul nivel (eng. *layer*) de dimensiunea 8193 care constă din două reprezentări a trăsăturilor (eng. *features*) a cadrelor procesate, care sunt de dimensiunea 4096 fiecare și o etichetă care indică relațiile dintre cadre. Cu ajutorul acestor date, rețeaua învață. Fiind o rețea siameză, învățarea se face cu ajutorul funcției *contrastive loss*. Aceasta determină cât de asemănătoare sau nu sunt cadrele încărcate în model pe baza distanței dintre trăsăturile acestora. Dacă cadrele sunt din același video, adică sunt o pereche pozitivă, funcția penalizează distanța mare dintre reprezentările trăsăturilor cadrelor, și recompensează distanța mică dintre ele. Dacă cadrele sunt din video-uri diferite, se calculează distanța dintre trăsături. Dacă distanța este mai mică decât un anumit prag, funcția returnează diferența dintre prag și distanță, dacă diferența este mai mare, funcția returnează 0. Astfel dacă distanța dintre două cadre din video-uri diferite este mai mică de un anumit prag, aceasta este penalizată. Funcția *contrastive loss* este și ea bazată pe funcția folosită de autorii lucrării originale care la rândul lor s-au bazat pe Hadsell et al. 2006 din articolul *Dimensionality Reduction by Learning an Invariant Mapping*<sup>9</sup>

Un pericol al antrenării, atât supervizate cât și a celei nesupervizate este *overfitting*-ul și *underfitting*-ul modelului. *Overfitting* este procesul în care un model învață foarte bine doar anumite trăsături ale setului de intrare, dar neglijează pe celelalte. *Underfitting* reprezintă procesul în care un model nu învață nimic.

O soluție pentru *underfitting* este să mărim setul de date. Însă această abordare nu e mereu posibilă. O alternativă ar fi ca datele de intrare să fie alterate în diferite moduri și apoi introdse în model pentru a fi antrenate. Această abordare ar putea fi o soluție, dar nu și în cazul când ai foarte multe date și putere de procesare limitată în timp sau spațiu. Neavând metode de a evita *underfitting*-ul, unica speranță ar fi ca numărul de iterații selectat să fie îndeajuns pentru ca modelul să învețe.

Soluțiile pentru *overfitting* sunt regularizarile L2 și Dropout. Dropout alege aleator un număr de rezultate și le face 0, astfel anulându-le. Cealaltă regularizare influențează rezultatele mici.

---

<sup>9</sup> <http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf>

Regularizarea L2 favorizează și generează valori mici pentru ponderi, astfel evitând erorile și diferențele mari pentru ponderile cu valori mari. Pentru antrenare am folosit Dropout cu rata de 0.5 și L2 cu descreșterea ponderilor (eng. *weight decay*) de 0.005.

## 2.5 Implementarea în Keras

Keras este o bibliotecă open-source ce reprezintă un API *high-level* pentru rețelele neuronale, scrisă în Python. Aceasta este capabilă să fie rulată având ca *backend* mai multe biblioteci de învățare automată, ca Tensorflow, Microsoft Cognitive Toolkit sau Theano. Pentru antrenare am folosit backend-ul Tensorflow.

Implementarea AlexNet-ului poate fi găsită ușor online. Adunând mai multe surse, consultând tutorialul din Coursera și lucrarea originală, am creat propria mea implementare a AlexNet-ului în Keras. Ceea ce m-a ajutat ulterior să ajung la arhitectura folosită în articol.

Pentru crearea arhitecturii modelului am folosit Sequential API din Keras. Am adăugat fiecare nivel necesar, având în Keras predefinite anumite tipuri de nivele. Pentru concatenarea a două nivele am folosit funcția `Concatenate` și am oferit ca parametri a acestei funcții un vector ce conținea eticheta ce reprezenta dacă perechea de video este pozitivă sau negativă și reprezentarea trăsăturilor celor două cadre procesate.

Keras-ul oferă posibilitatea de a avea o funcție custom pentru învățarea modelului. Este exact ceea ce aveam nevoie pentru a crea funcția *contrastive loss*. Funcțiile custom trebuie să conțină 2 parametri: `y_true` și `y_pred`. `y_true` reprezintă etichetele datelor de intrare, adică ce reprezintă acestea (eng. *ground truths*). `y_pred` reprezintă ceea ce a prezis modelul. Deoarece antrenarea se face nesupervizat nu avem nevoie să folosim `y_true`.

Deoarece funcția necesită o condiție, natural m-am hotărât să implementez soluția printr-o condiție `if`. Cum am spus, ultimul nivel al modelului are dimensiunea de 8193. Naiv am presupus că această dimensiune reprezintă dimensiunea unui `numpy.array`, însă după mai multe ore am aflat că nu aveam dreptate. Keras-ul fiind bazat pe Tensorflow, la inițializare crează în spate un graf pe care Tensorflow-ul îl compilează și apoi îl rulează. Ce se află ca intrare în funcția custom *contrastive loss*, nu era un `numpy.array` dar era un obiect `Tensor` din biblioteca Tensorflow. Acest obiect, deși deține și el o dimensiune ca un `numpy.array`, se comportă foarte diferit. E mai mult o funcție ce are un input și un output, care reprezintă un nod într-un graf decât un simplu vector. Deci nu puteam face condiționarea direct pe obiect. Astfel am încadrat condiția de evaluare în însăși funcția matematică. Am extras valorile trăsăturilor celor două cadre și etichetel folosite în trei obiecte

de tip `Tensor`. Am ajuns la o funcționalitate condiționată implementând o mască asupra funcției matematice. Implementarea funcției *contrastive loss* poate fi vizualizată în **Figura 3**.

```
def contrastive_loss_function(y_true, y_pred):
    # y_true = label, target tensor, not used in our case
    # y_pred = prediction, model output tensor

    # just so pep won't annoy me
    _ = y_true

    # the flag for the same video or not is concatenated for the prediction and the both encodings
    # WARN: these are tf.Tensor not np.array
    diff_video_frame_flag = y_pred[0][0]
    query_encoding = y_pred[0][1:4097]
    compared_encoding = y_pred[0][4097:]

    euclidian_distance = keras_backend.sqrt(
        keras_backend.sum(keras_backend.square(query_encoding - compared_encoding), axis=-1, keepdims=True))

    alt_distance = keras_backend.maximum(MARGIN_DELTA - euclidian_distance, 0)

    # simulate the if condition, one side will always be zero
    return (diff_video_frame_flag * euclidian_distance) + ((1 - diff_video_frame_flag) * alt_distance)
```

**Figura 3.** Implementarea funcției *contrastive loss*

Am constatat la început că am nevoie să salvez modelul. Keras oferă posibilitatea de a salva modelul sau doar ponderile acestuia în orice moment al antrenării. Acest lucru este foarte folositor luând în considerare că antrenarea necesită mult timp de procesare, și în caz că apare vreo eroare în codul sursă sau mașina în care este rulat modelul întâmpină o eroare, să nu fie pierdute datele antrenate deja. Dar ce anume ar fi bine să fie salvat? Dacă salvez doar ponderile, fișierul de salvare va fi mult mai mic, însă la încărcarea fișierului salvat voi fi nevoit să recreez toată arhitectura modelului și astfel voi pierde starea optimizatorului. Astfel am optat pentru salvarea modelului în întregime.

Pentru a salva tot modelul creat, care face parte atât din nivelele create cât și ponderile antrenate în urma învățării, este folosită funcția `save` a obiectului `Model` din Keras. Rezultatul acestei salvări este un fișier de tip `HDF5`. Iar pentru încărcarea modelului salvat, se poate folosi funcția `get_saved_model` din același obiect `Model`, dându-i ca parametru calea către fișierul salvat. Având o funcție custom, trebuie oferită explicit funcției de încărcare a modelului, altfel o eroare este ridicată.

Implementarea autorilor implică folosirea conceptului de *mini-batch*. Acesta actualizează ponderile modelului nu după fiecare input, dar la un anumit interval, și anume la dimensiunea *mini-batch*-ului. Modelul învață în acest *mini-batch*, actualizează ponderile “local” și apoi când s-a



terminat *mini-batch*-ul actualizează ponderile globale. Autorii au folosit pentru dimensiunea *mini-batch*-ului 120 de perechi de cadre pentru antrenarea modelului lor.

Keras-ul conține 3 funcții de antrenare: `fit`, `fit_generator` și `train_on_batch`. Funcția `fit` presupune că întreg setul de date de antrenare poate să încapă în memorie, ceea ce în cazul nostru nu e posibil. Funcția `fit_generator` necesită un generator în python care ofera *mini-batch*-urile, astfel eliminând necesitatea stocării a întregii mulțimi de date în memorie. Funcția `train_on_batch` acceptă un singur *mini-batch*, execută *Backpropagation*-ul și apoi actualizează modelul.

Pentru evaluarea modelului am folosit funcția `predict_on_batch` care preia o singură dată de intrare. Precum input-ul modelului conține două cadre și un label, iar pentru testare avem doar un singur cadru, trebuie să oferim modelului date false (`numpy.array`-uri ce conțin doar 0-uri), pentru că oricum la testare modelul nu mai învață. Pentru predicția corectă a modelului, datele de intrare trebuie preprocesate din nou folosind funcțiile `expand_dims` din `numpy` și `preprocess_input` din Keras, pentru a adapta cadrul pentru inputul modelului, altfel toate predicțiile vor returna 0. Predicția modelului este un `numpy.array` de dimensiunea 8193, din care trebuie să extragem trăsăturile specifice cadrului de testare. Apoi având lista cu reprezentările tuturor cadrelor din video-urile de testare evaluăm modelul. Detaliile evaluării se pot găsi în **Capitolul 5. Evaluarea modelului și vizualizarea rezultatelor**.

Funcția `fit_generator` execută exact ceea ce am eu nevoie, dar pe lângă antrenarea modelului aveam nevoie și să îl salvez, astfel aveam nevoie de o funcție care să îmi poată oferi mai mult control asupra învățării modelului. Aceasta era oferită de funcția `train_on_batch`, pe care am folosit-o.

Acum trebuia să creez un generator de *mini-batch*-uri. O posibilă soluție ar fi fost să folosesc funcția `flow_from_directory` din obiectul `ImageDataGenerator` din Keras, care crează *mini-batch*-urile direct dintr-un director. Am fi putut folosi această funcție dacă am fi avut toate cadrele procesate într-un singur director. Dar astfel încât fiecare video are nevoie de propriul director trebuia să găsesc o alternativă cu ajutorul căreia puteam să parcurg ușor directoarele și să creez *mini-batch*-urile.

Am decurs la o parcurgere manuală a directoarelor ce reprezentau cadrele preprocesate ale video-urilor din mulțimea de date și crearea *mini-batch*-urilor. Am creat codul să fie ușor modificabil. Partea negativă când faci ceva manual e că presupui multe lucruri fără ca să le știi sigur. Așa am presupus că `train_on_batch` primește *mini-batch*-ul sub forma unui vector, în care

fiecare element reprezintă un *mini-batch*. În realitate nu era așa. Funcția aștepta un vector ce conținea fiecare input din model într-un vector separat.

Pentru citirea imaginilor din format `jpg` într-un `numpy.array` de forma `(227, 227, 3)` am folosit la început funcția `load_img` din biblioteca `keras.preprocessing.image`, care încarcă o imagine în formatul PIL (*Python Imaging Library*) iar apoi am convertit-o în forma dorită folosind funcția `img_to_array` din aceeași bibliotecă.

Având lista de video-uri pe care trebuie procesată, atât pentru antrenare cât și pentru evaluare, am nevoie să iterez prin directoarele cadrelor procesate și să creez *mini-batch*-ul ca un *buffer*. În liste dețineam calea relativă a video-urilor, și cu ajutorul ei găseam video-ul ce conținea cadrele necesare procesării.

Pentru a rula codul Keras pe placa video, nu este nevoie de nici o setare în cod, astfel încât codul va rula automat pe placa video dacă găsește una pe care poate rula.

## Capitolul 3. Preprocesarea video-urilor

### 3.1 Mulțimea de date folosită

Pentru învățarea unui model de rețele neuronale, fie supervizate sau nu, este necesară o anumită mulțime de date pe baza căreia modelul va învăța. Datele pot fi de orice tip, fie text, imagine, video etc. Această mulțime de date poate fi selectată manual pentru a satisface cerința unei anumite probleme sau poate fi folosită o mulțime de date deja creată în urma unor anumite studii care se axează pe un anumit domeniu. Multe mulțimi de date pot fi găsite online și descărcate gratuit, astfel încât majoritatea cercetătorilor le fac publice după cercetare. Exemple de mulțimi de date publice: MNIST, ImageNet, CIFAR-10 etc.

Astfel încât acumularea unei mulțimi de date specifice unei probleme ar lua mult prea mult timp am decis să folosesc o mulțime de date deja existentă și care conține datele de care am nevoie. Mulțimea de date folosită pentru antrenarea modelului este UCF101<sup>10</sup>. UCF101 este o mulțime de date pentru recunoaștere de acțiuni ce conține 13,320 de video-uri colectate din YouTube și clasificate în 101 categorii. Aceste categorii pot fi împărțite în 5 tipuri: 1) Interacțiunea om-obiect 2) Mișcarea corpului 3) Interacțiunea om-om 4) Acțiunea de a cânta la un instrument muzical 5) Acțiuni sportive

Fiecare categorie conține 25 de grupe. O grupă de video-uri este în esență un video lung dedicat unei acțiuni ce a fost împărțit în 4-7 video-uri mai scurte, cu lungimi ce variază între 5 și 10 secunde. Video-urile din aceeași grupă pot să împărtășească aceleași trăsături ca: fundal asemănător, același punct de vedere (eng. *point of view*) etc. Această abordarea este binevenită astfel încât ușurează procesul de preprocesare.

Pe lângă arhiva ce conține mulțimea de date, pe site-ul mulțimii de date, este publicată o altă arhivă ce conține 3 împărțiri diferite a mulțimii pentru antrenare și testare. Fiecare împărțire conține în jur de 9500 de video-uri de antrenare și aproximativ 3500 de video-uri de testare. Aceste împărțiri sunt exprimate ca și liste. Atât listele pentru antrenare și pentru testare folosesc caile relative către fiecare video ce trebuie procesat. La listele de antrenare este prezentă și eticheta (eng. *label*) acestui video ce reprezintă categoria din care face parte. Eticheta aceasta este reprezentată ca un număr, iar relația dintre aceste numere și categoriile din mulțimea de date este găsită într-un alt fișier, de asemenea prezent în arhivă.

La început am dorit să reproduc cât de mult posibil ce au făcut autorii articolului original, astfel am decis să iau același pachet de antrenare/testare de video-uri folosit de ei. Acesta constitue

---

<sup>10</sup> <https://www.crcv.ucf.edu/data/UCF101.php>

din 9537 de video-uri pentru învățare și 3783 pentru testare. Însă, precum este detaliat în **Capitolul 4. Antrenarea modelului și mașina virtuală**, am fost nevoit să reduc dimensiunea mulțimii de date, din cauza lipsei de putere de procesare. Astfel din 101 categorii am scăzut la doar 10, din 9537 de video-uri pentru învățare am scăzut la 1029, iar din 3783 de video-uri de testare am scăzut la 409.

Având libertatea de a alege tipul de categorie, am selectat *Acțiuni sportive*. Categoriile selectate sunt:

1. Tragerea cu arcul (eng. Archery)
2. Biliard (eng. Billiards)
3. Bowling
4. Lovitură de golf (eng. Golf Swing)
5. Aruncarea cu ciocanul (eng. Hammer Throw)
6. Curse de cai (eng. Horse Race)
7. Canotaj (eng. Rowing)
8. Schi (eng. Skiing)
9. Lovitură de pedeapsă în fotbal (eng. Soccer Penalty)
10. Lovitură de tenis (eng. Tennis Swing)

### 3.2 Algoritmul de preprocesare

Modelul are ca input două cadre, sau mai bine spus două reprezentări a două cadre. Mulțimea de date este formată doar din video-uri, deci apare necesitatea unui algoritm de preprocesare a video-urilor în cadre separate pentru a putea fi introduse în model pentru antrenare. Sunt două metode de abordare a preprocesării: preprocesarea în timpul antrenării și preprocesarea independentă de antrenare.

Preprocesarea în timpul antrenării constă din procesarea datelor doar atunci când este necesară. Astfel parcurg lista de video-uri pe care trebuie să le ofer modelului, extrag cadrele din video, le introduc în model, apoi continui cu următorul video. Această abordare ar fi potrivită dacă ar exista o limită de memorie de stocare, astfel în memorie vor fi doar cadrele unui singur video. Partea negativă a acestei abordări este că dacă modelul eșuează, trebuie să reia toată preprocesarea din nou. Dar pe lângă asta, precum a fost menționat și în **Capitolul 2. Arhitectură și implementare**, modelul necesită perechi de cadre atât pozitive, din același video, cât și perechi de cadre negative, din cadre diferite. Asta ar însemna că pentru fiecare cadru negativ ar trebui să preprocesăm un video aleator în plus pe lângă cel care se antrenează. Și precum la fiecare a 5-a pereche una trebuie să fie

negativă, vom preprocesa multe video-uri doar pentru un singur cadru aleator. Ceea ce ar slăbi performanța antrenării modelului. Aceasta poate fi evitată folosind cealaltă metodă de preprocesare.

Preprocesarea independentă de antrenare constă din preprocesarea a tuturor video-urilor înainte de antrenare și apoi parcurgând lista cu video-uri pe care trebuie să le învețe modelul, oferind modelului doar calea absolută către cadrele strict necesare învățării. Astfel, separăm acțiunea de învățare a modelului cu cea de preprocesare a input-ului. Abordarea aceasta mai aduce un beneficiu. O dată ce video-urile au fost procesate, toate cadrele rămân acolo, și dacă dorim să rerulăm modelul, nu avem decât să rerulăm doar modelul, nu și preprocesarea. Astfel am salvat și timp.

Odată ce am ales modul de execuție a preprocesării am rulat niște teste să văd câtă memorie este necesară pentru toate cadrele procesate. Am rulat preprocesarea pe laptop-ul meu și am avut următoarele rezultate. Pentru 30 de video-uri din care am extras toate cadrele, dimensiunea directorului a crescut la 73.9MB în 19.4 secunde. Considerând că avem peste 9500 de video-uri, dimensiunea ar fi crescut la aproximativ 23.5GB, iar din punct de vedere al timpului, pe mașina mea locală, ar fi luat 1.7 ore. Timpul rulării este prezentat doar ca un punct de reper, în realitate preprocesarea a fost rulată pe mașina virtuală, unde aveam mult mai multă putere computațională.

Realizând că mi-ar fi luat prea mult timp și spațiu pentru preprocesare, am dorit să caut o soluție mai eficientă. Fiecare video conține între 100 și 200 de cadre, dar nu avem nevoie de fiecare cadru pentru antrenare. Continuitatea între cadre nu asigură informație nouă. Pixelii a două cadre succesive nu diferă foarte mult, și astfel nu contribuie la învățarea noilor trăsături (eng. *features*). La început doream să extrag un număr fix de cadre aleatoare din fiecare video. Această abordare ar fi adus anumite dezavantaje, astfel încât nu extrăgeam într-un mod consistent cadrele din video, și aveam riscul de a extrage doar cadrele de la o porțiune a video-ului sau să repet cadrele deja procesate. De aceea am procesat din fiecare 10 cadre a video-ului unul singur. Acest număr a fost ales în urma observării empirice cât și consultând codul sursă confuz din articolul original. Schimbând algoritmul inițial de preprocesare, să salveze doar al 10-lea cadru din video, am obținut rezultate mult mai bune. Rulând același test de 30 de video-uri am obținut, 9.05MB dimensiunea directorului într-un timp de 4 secunde. Ceea ce reprezintă aproximativ o îmbunătățire de 5 ori în timp și aproximativ 8 ori din punct de vedere al spațiului. Preprocesarea în mașina virtuală, pentru toate video-urile, a constituit 4122.0793914794 secunde sau 1.145 ore. După ce am redus numărul de categorii, timpul execuției a scăzut la 372 de secunde care sunt 6.2 minute.

Pentru a citi cadrele dintr-un video am folosit biblioteca OpenCV<sup>11</sup> și anume obiectul `VideoCapture` și funcția acestuia `read()` care returnează cadrul următor din video și un flag

---

<sup>11</sup> <https://opencv.org/>

care indică dacă a fost extras cu succes sau nu. Inițial am creat un algoritm care avea nevoie de numărul total de cadre dintr-un video. Pentru aceasta, extrăgeam din obiectul `VideoCapture` creat proprietatea `CAP_PROP_FRAME_COUNT`. Dar după ce am rulat modelul în mașina virtuală, am depistat o eroare, mai degrabă o consecință nedorită. Proprietatea menționată anterior nu oferă numărul exact de cadre, ci mai degrabă oferă doar o aproximare a acestui număr. Având nevoie de numărul exact, trebuia să caut o altă soluție.

O altă soluție a fost să nu mă bazez deloc pe lungimea video-ului, dar să creez o fereastră de două cadre cu care parcurg video-ul, unul pentru procesarea (*query frame*) iar celălalt pentru verificare (*next frame*). Cât timp erau două cadre continue în acea fereastră, verificam dacă trebuie să salvez cadrele, adică când numărul de cadre parcurse era divizibil cu 10, parcurgând astfel al fiecarele 10 cadru. Iar când această condiție se îndeplinea, verificam dacă trebuie să salvez cadrul de verificare sau nu. După consultarea din nou a codului sursă original, am decis să iau 4 din 5 perechi de cadre să fie pozitive, adică din același video, și a 5-a să fie negativă, adică un cadru aleator dintr-un video aleator dintr-o categorie aleatoare. Astfel condiția pentru salvarea cadrului de verificare era ca numărul de cadre salvate să nu fie divizibil cu 5, adică a fiecare a 5-a pereche de cadre nu salvează cadrul de verificare, pentru că este perechea negativă. Astfel optimizând timpul de preprocesare.

Cadrele sunt salvate în felul următor. Mai întâi fiecare categorie își are propriul director. Apoi în acest director fiecare video inițial își are propriul director în care se conțin toate cadrele extrase din acest video. Cadrele care vor fi procesate au următoarea denumire: `query_frame_XXXXX.jpg`, unde `XXXXX` reprezintă numărul cadrului procesat. Cadrul imediat următor din video, care este necesar pentru perechea pozitivă din model, se află în același director având denumirea de `next_frame_XXXXX.jpg`, unde `XXXXX` este numărul de cadru căruia îi este asociat.

Modelul are ca formă de intrare o dimensiune fixă, și anume 227 pe 227 pixeli. Fiecare video are dimensiuni diferite, deci fiecare cadru din fiecare video trebuie redimensionat. Preferabil aceasta trebuie făcută în timpul preprocesării, ca în final să avem din toată mulțimea de video-uri doar cadrele strict necesare pentru învățarea modelului și ajustate pentru model. Astfel în algoritmul de preprocesare înaintea salvării cadrului, îl redimensionez folosind `cv2.resize`, iar apoi îl salvez ca fișier `jpg` folosind `cv2.imwrite`. Astfel am preprocesat un video din formatul `avi` în cadre în formatul `jpg`.

## Capitolul 4. Mașina virtuală și antrenarea modelului

### 4.1 Google Cloud Platform

Pentru învățarea oricărei rețele neuronale este necesară putere computațională. Pentru o rulare rapidă, și o învățare relativ rapidă a modelului, am decurs la soluții cloud, și anume Google Cloud Platform<sup>12</sup>. La înregistrarea unui cont ai disponibil \$300 credit valabili 12 luni. Cu acest credit poți folosi anumite servicii pe platforma lor, pentru care de altfel ar fi trebuit să plătești. Unul din ele, și ceea ce am folosit eu, a fost *Computing Engine*, adică o mașină virtuală.

La început doream să configurez o mașină virtuală pe cât mai mult timp posibil, adică cât mai ieftin dar pe care să o pot folosi. Cu aceasta în gând am configurat o mașină virtuală foarte slabă, având doar 2 procesoare, fără placă video și o memorie RAM de 6GB la aproximativ \$75 pe lună, ceea ce mi-ar fi oferit în jur de 3 luni de operabilitate. Când am intrat pe mașina virtuală, folosind tool-ul integrat în Windows: Remote Desktop Connection, rapid mi-am dat seama că nu o să fie de ajuns, și m-am gândit la o altă soluție.

În loc să maximizez timpul folosit de mașina virtuală, ar fi mai bine să maximizez resursele oferite de Google Cloud Platform. Pentru aceasta am dorit să configurez o mașină virtuală cu o placă video. La început doream iar să maximizez timpul, dar am observat rapid că prețul pentru mașinile virtuale cu plăci video sunt considerabil mai scumpe decât cele fără, și pe lângă asta platforma oferea doar plăci de video performante.

Platforma nu îți permite să folosești o placă video dacă nu îți activezi contul. Adică trebuie să treci de la un cont gratuit (eng. *free trial*) la un cont plătit (eng. *paid account*). Dacă vrei să creezi o mașină virtuală cu o placă video trebuie să explici motivul pentru care vrei să o folosești. În urma cererii ei îți trimit un email în care îți explică că cererea este procesată în timp de 2 zile lucrătoare, și dacă e urgent poți să le trimiți un reply să explici de ce e urgent. Am depus cererea seara și a doua zi dimineața a fost aprobată.

După o analiză a posibilităților de configurare a noii mașini virtuale, am ajuns la o concluzie decisivă. Mașina pe care am rulat codul arată așa: 4 procesoare, 15 GB RAM, placă video NVIDIA TESLA V100 și rula Windows Server 2016. Toate la prețul de aproximativ \$1200 pe lună sau \$2.04 pe oră. Partea bună din asta este că ești taxat doar pe ceea ce consumi. Adică aveam oportunitatea de a calcula câte iterații posibile pot face în mașina mea virtuală până când nu o să mai am credit. Și exact asta am făcut.

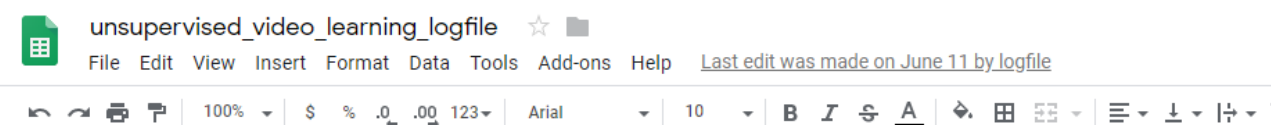
---

<sup>12</sup> <https://cloud.google.com/>

## 4.2 API-uri Google

Rulând codul pe o mașină virtuală asupra căreia nu am acces, am vrut să fiu sigur că pot vedea progresul având acces doar la internet și să salvez modelul la un anumit interval. Pentru a asigura aceste două funcționalități am folosit Google Sheets API și Google Drive API.

**Google Sheets API**, l-am folosit pentru scrierea informațiilor relevante într-o foaie de stil (eng. *spreadsheet*). În timpul învățării scriam câte o linie din tabel la fiecare iterație. Informația relevantă includea 4 lucruri: când a fost scrisă linia în tabel (eng. *timestamp*), ajustată la fusul orar din România, mașina virtuală fiind în Iowa, SUA, câte iterații au trecut, câte ore au trecut de la începutul antrenării, și dacă a fost salvat sau nu modelul. O porțiune din spreadsheet este prezentată în **Figura 4**. Înserând aceste date într-un spreadsheet ce poate fi accesat de mine oricând, puteam vedea progresul învățării modelului de oriunde aveam acces la internet.



	A	B	C	
136	2019-06-08 22:34:14.170697	58.058537387499996	CURRENT NO. EPOCH: 428	MODEL SAVED: False
137	2019-06-08 22:26:11.140015	57.924362198055555	CURRENT NO. EPOCH: 427	MODEL SAVED: False
138	2019-06-08 22:18:06.968271	57.78987004694445	CURRENT NO. EPOCH: 426	MODEL SAVED: False
139	2019-06-08 22:10:03.437362	57.65555590555555	CURRENT NO. EPOCH: 425	MODEL SAVED: False
140	2019-06-08 22:01:58.717895	57.52091160916667	CURRENT NO. EPOCH: 424	MODEL SAVED: False
141	2019-06-08 21:53:54.859057	57.38650637638889	CURRENT NO. EPOCH: 423	MODEL SAVED: False
142	2019-06-08 21:45:51.265526	57.25217484	CURRENT NO. EPOCH: 422	MODEL SAVED: False
143	2019-06-08 21:37:46.859389	57.11761757972223	CURRENT NO. EPOCH: 421	MODEL SAVED: False
144	2019-06-08 21:29:41.108750	56.98268684666667	CURRENT NO. EPOCH: 420	MODEL SAVED: True
145	2019-06-08 21:21:06.671822	56.8397877	CURRENT NO. EPOCH: 419	MODEL SAVED: False
146	2019-06-08 21:13:03.312534	56.705521231111106	CURRENT NO. EPOCH: 418	MODEL SAVED: False
147	2019-06-08 21:05:01.374154	56.57164945888889	CURRENT NO. EPOCH: 417	MODEL SAVED: False
148	2019-06-08 20:56:58.374680	56.43748293833333	CURRENT NO. EPOCH: 416	MODEL SAVED: False
149	2019-06-08 20:48:55.577752	56.303372680555555	CURRENT NO. EPOCH: 415	MODEL SAVED: False
150	2019-06-08 20:40:53.140757	56.16936240416667	CURRENT NO. EPOCH: 414	MODEL SAVED: False
151	2019-06-08 20:32:50.452163	56.03528223916667	CURRENT NO. EPOCH: 413	MODEL SAVED: False
152	2019-06-08 20:24:47.155835	55.90103325916667	CURRENT NO. EPOCH: 412	MODEL SAVED: False
153	2019-06-08 20:16:42.921425	55.76652370083333	CURRENT NO. EPOCH: 411	MODEL SAVED: False
154	2019-06-08 20:08:39.031196	55.63210974833333	CURRENT NO. EPOCH: 410	MODEL SAVED: True

**Figura 4.** Spreadsheet-ul folosit pentru înregistrarea datelor relevante pe parcursul antrenării modelului

Pe lângă salvarea informației relevante în timpul antrenării, am setat ca fiecare eroare întâmpinată să fie afișată, prin împachetarea funcției principale într-un bloc `try/catch`. Astfel în caz că apare o eroare în timpul rulării, o voi vedea și o să pot să intru pe mașina virtuală și să o corectez. Precauția aceasta s-a dovedit a fi folositoare.

La fel, în același spreadsheet, am afișat și progresul de testare a modelului și rezultatul final. Astfel în acest spreadsheet, poate fi observat tot procesul învățării și evaluării modelului.



**Google Drive API**, l-am folosit pentru salvarea fișierului HDF5 ce reprezenta modelul salvat, pe platforma de stocare a datelor în cloud Google Drive. La fiecare a 10-a iterație salvam modelul și îl trimiteam în contul meu Drive. Am creat modulul relativ generic astfel încât să pot trimite orice tip de fișier așa avea nevoie. A fost folositoare această abordare, astfel încât am trimis atât modelul salvat cât și imaginile ce reprezentau vizualizarea datelor în format `jpeg`.

Fișierul HDF5 are o dimensiune constantă de 230MB, și lua în medie un minut să fie încărcat pe server. Nu aveam nevoie de fiecare salvare a modelului. În primul rând, pentru că nu dispuneam de memoria necesară pe platforma Drive și în al doilea rând ultima salvare a fișierului conține cele mai relevante date, conținând progresul antrenării modelului pentru cele mai multe iterații. Google Drive API îți oferă posibilitatea să încarci o versiune nouă a unui fișier dacă deții `FileId`-ul acestui fișier. Înainte de salvarea modelului pe Drive, verificam dacă dețin un `FileId`. Dacă exista, atunci actualizam fișierul existent, dacă nu, cream unul nou. Request-urile pentru aceste acțiuni sunt diferite. În urma creării unui fișier nou, funcția apelată din API returnează `FileId`-ul nou creat. Versiunile se șterg după 30 de zile sau după ce 100 de versiuni au fost salvate.

Pentru a putea utiliza API-urile pe parcursul a mai multor zile, trebuia să mă asigur că folosesc un token valid de fiecare dată când le folosesc. Prima intuiție a fost să încerc să mă logez în fiecare zi în contul Google pe mașina virtuală și astfel să actualizez token-ul. Imediat mi-am dat seama că nu e o idee bună, pentru că e predispusă la eroare. Astfel trebuia să găsesc o alternativă. Token-ul oferit de API-ul Google Drive era în realitate un refresh token, ceea ce permitea actualizarea acestuia fără ca utilizatorul, adică eu, să fac autentificarea în sine. Actualizarea token-ului am făcut-o direct în codul sursă. Înainte de executarea API-ului verificam dacă token-ul e valid, dacă nu era valid îl actualizam și continuam execuția. Asta a permis folosirea API-ului Google Drive în tot timpul antrenării fără nici o eroare.

### 4.3 Setarea mașinii virtuale

Odată ce am pornit mașina virtuală următorul pas era să instalez toate resursele necesare mediului de dezvoltare. După ce am instalat Python 3.6.5<sup>13</sup> și editorul PyCharm Community Edition<sup>14</sup>, am continuat cu instalarea bibliotecilor utilizate folosind sistemul de management de pachete pentru python `pip`. Pentru a instala un pachet cu o anumită versiune am folosit comanda:

```
python -m pip install <denumirea_pachetului>==<versiunea_dorita> --user
```

---

<sup>13</sup> <https://www.python.org/downloads/release/python-365/>

<sup>14</sup> <https://www.jetbrains.com/pycharm/download>

Bibliotecile folosite, versiunile acestora și scopul acestora în mediul de dezvoltare sunt următoarele:

- tensorflow 1.12.1 (backend-ul folosit de Keras)
- keras 2.1.6 (implementarea modelului)
- opencv-python 4.1.0 (preprocesarea video-urilor)
- pillow 6.0.0 (citirea și stocarea imaginilor în vectori 3D)
- gspread 3.1.0 (conectarea către API-ul Google Sheets)
- authlib 0.11 (permite actualizarea token-ului pentru gspread)
- google-api-pyhon 1.7.9 (conectarea către API-ul Google Drive)
- google-auth-httpplib2 0.0.3 (conectarea către API-ul Google Drive)
- google-auth-oauthlib 0.4.0 (conectarea către API-ul Google Drive)
- scikit-learn 0.21.1 (aplicarea algoritmului K-means și tSNE)
- matplotlib 3.1.0 (vizualizarea rezultatului final)

Pe lângă toate pachetele instalate aveam nevoie și de resursele necesare programării pe placa video. Pe site-ul oficial Tensorflow<sup>15</sup> sunt afișate resursele necesare pentru instalarea driverelor necesare perimterii rulării framework-ului pe placa video. Deși link-urile oficiale oferă resuresele potrivite pentru ce ai nevoie, nu oferă și versiunile care chiar funcționează. Natural aștepți să meargă ultima versiune cel mai bine, dar din păcate nu e cazul când e vorba de programare pe placa video.

După instalarea destul de îndelungată a resurselor Nvidia și CUDA, când încerci să verifici prin Tensorflow ce placă video e valabilă, primești erori foarte neintuitive, erori direct în codul executat pentru că nu găsește anumite resurse. După ce mi-a luat mult timp cătuarea unei soluții, am găsit un comentariu pe GitHub Issues<sup>16</sup> care arăta o configurație care mergea. Am instalat-o la mine și a mers. E foarte neintuitiv pentru că sunt mai multe componente, fiecare cu versiunile sale, și dacă nu selectezi versiunile compatibile, nu o să funcționeze și nici nu o să ai eroarea clară care ar indica sursa problemei.

Configurația care a funcționat pentru mașina virtuală folosită a fost următoarea:

- CUDA 9.0
- Tensorflow 1.12.0
- CUDNN 7.4.1.5

---

<sup>15</sup> <https://www.tensorflow.org/install/gpu>

<sup>16</sup> <https://github.com/tensorflow/tensorflow/issues/22794#issuecomment-442039709>

Keras-ul dacă detectează o placă video pe mașina pe care este rulat, rulează pe placa video automat în spate fără a fi configurat explicit. Confirmarea vine la inițializarea modelului:

```
[4] 0: N
[97] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 14957 MB memory) -> physical GPU (device: 0, name: Tesla V100-SXM2-16GB, pci bus id: 0000:0
```

#### 4.4 Alegerea numărului de categorii și numărului de iterații

Precum am spus aveam \$300 credit gratuit în contul Google Cloud Platform. După instalarea drivere-lor necesare programării pe placa video, după preprocesarea tuturor video-urilor din mulțimea de date am rulat modelul o singură dată pentru a monitoriza timpul rulării unei iterații. La început voiam să refac același experiment rulat în articolul original, dar rapid mi-am dat seama că nu e posibil.

Nu am rulat o iterație completă dar având unele date ca punct de reper am putut calcula cât timp am nevoie pentru o iterație completă. Pentru învățarea a doar 7 categorii dintr-o iterație, modelului i-a luat 10 minute. 101 de categorii ar însemna  $14.5 \times 10 \text{ min} = 145 \text{ min} = 2.5 \text{ ore}$ . Luând în considerare că la momentul rulării mai aveam \$284 rămași în credit, și configurația mașinii virtuale costa \$2.04 pe oră, ar însemna că voi putea rula antrenarea modelului în jur de 130-140 de ore pe mașina virtuală. Orele acestea ar însemna în jur de 54 de iterații pentru toate 101 categorii. Modelul nu ar învăța aproape nimic considerând dimensiunile mulțimii de date sau dacă ar învăța ceva ar fi complet nesemnificativ. Pe lângă asta, nu aș putea face o comparație cu rezultatul articolului original, pentru că autorii au rulat modelele lor pentru 30.000 de iterații. Ceea ce aș fi putut face eu ar reprezenta doar 0.18% din ceea ce au făcut ei, deci virtual nimic.

Unica soluție în acest caz a fost să micșorez mulțimea de date folosită pentru antrenare. Dacă aș fi ales 25 de categorii în loc de 101, o iterație ar lua aproximativ 30 de minute. Timpul acesta ar permite antrenarea modelului pentru aproximativ 270 de iterații. Aceasta prezintă o anumită îmbunătățire față de configurația precedentă, dar numărul de iterații nu inspiră încredere. Deci am decis să scad și mai mult, de data aceasta la 10 categorii, deci de 10 ori mai mică mulțimea de date. Presupunând că ar lua aproximativ 10-15 minute o iterație, am fi avut în total 500 de iterații. În loc să avem un model care învață slab o mulțime de date mare, am putea avea rezultatele mult mai bune pe un model care învață bine sau relativ bine o mulțime mică de date.

După alegerea numărului de categorii optim pentru configurația mașinii virtuale am selectat categoriile dorite pentru antrenare, care au fost prezentate în **Capitolul 3. Preprocesarea video-urilor**. Am recreat listele de video-uri pentru antrenare și testare și am rulat modelul pentru o iterație. Aceasta mi-a luat 0.1334849772 ore, ceea ce reprezintă 8 minute. Luând în considerare din nou cât credit aveam în cont, am calculat din nou cât timp aș avea disponibil pentru antrenare. Acesta a fost de aproximativ 139 de ore, ceea ce reprezintă 5.79 zile sau 8340 minute, adică 1000 de iterații. Ceea

ce a reprezentat un timp mult mai bun față de prezicerea inițială. Astfel, inițial am decis să rulez antrenarea pentru 1000 de iterații, dar precum o să fie detaliat ulterior am fost nevoit să o scad la 500.

Precum e descris în **Capitolul 2. Arhitectură și implementare**, modelul inițial era implementat pe baza conceptului de *mini-batch*. Însă având o mulțime de date mult mai mică decât cea inițială, dispare optimizarea adusă de această abordare. Deci am fost nevoit să scad dimensiunea *mini-batch*-ului la 1. Modificarea a fost necesară doar într-un singur loc, astfel încât *mini-batch*-ul era creat dinamic.

## 4.5 Antrenarea modelului și problemele eventuale

După ce am setat tot mediul de dezvoltare, am rulat preprocesorul pe mulțimea de date, m-am logat în conturile Google pentru autentificarea API-urilor, am rulat o singură iterație completă pentru a verifica dacă nu apar erori. Procesul de antrenare constituie din rularea aceluiași set de cod de mai multe ori. Astfel dacă o iterație completă nu aruncă erori, pe tot procesul antrenării nu ar putea crea probleme. De asemenea am testat API-urile după autentificare și asigurându-mă că nu există potențiale erori am început antrenarea efectivă.

De la început mergea bine, dar după o oră mașina virtuală s-a oprit, sau mai bine spus nu mai afișa nimic în spreadsheet-ul meu. Am intrat pe mașina virtuală și am depistat că eroarea vine de la API-ul Google Sheets. Presupun ca sesiunea expira fix într-o ora, sau token-ul nu mai era valabil, în orice caz am modificat biblioteca folosită pentru autenticitatea API-ului și am rerulat antrenarea de la început. Această mică inconveniență m-a costat \$40, ceea ce a m-a impus să scad numărul de iterații de la 1000 la 750.

Dupa 58 de ore de învățare continuă, o eroare de conexiune (internal 500) a API-ului Google Sheets a fost aruncată seara. Am observat-o în spreadsheet dimineața, și în tot timpul parcurs mașina virtuală continua execuția astfel am mai pierdut din nou credit. Pentru momente din astea am implementat salvarea modelului. Eroarea a fost depistată cand modelul se afla la a 428-a iterație și mai aveam în cont \$58.12. Tot ce rămânea de făcut era să rotunjesc antrenarea la 500 de iterații. Ultimul model a fost salvat după a 420-a iterație, ceea ce însemna că mai am 80 de iterații de antrenat, care ar constitui aproximativ \$20. Am intrat din nou în mașina virtuală, am schimbat numărul de iterații, am încărcat modelul salvat și am rerulat antrenarea.

În total modelul a fost antrenat pentru aproximativ 68 de ore continue, adică aproximativ 2.8 zile.

## Capitolul 5. Evaluarea modelului și vizualizarea rezultatului

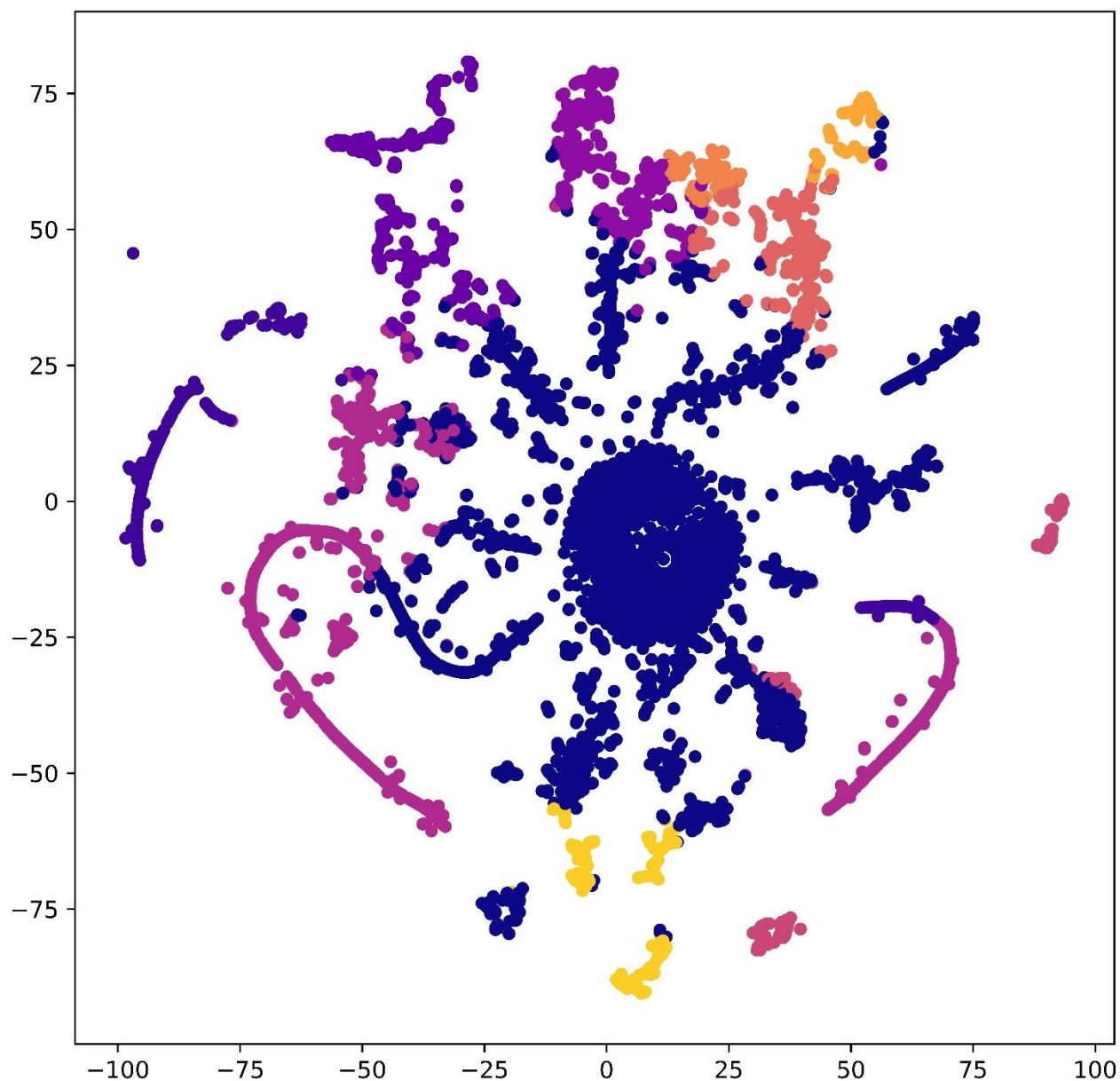
După antrenarea modelului, natural ar trebui să îl evaluăm. Fiind vorba de o antrenare nesupervizată nu avem cum să evaluăm modelul folosind metodele tradiționale de evaluare. După ce am antrenat modelul, am introdus cadrele video-urilor de testare și am acumulat toate reprezentările trăsăturilor (eng. *features*) acestora într-o listă.

Apoi pe această listă am aplicat algoritmul K-means din biblioteca `scikit-learn` modulul `cluster`, care returna indexul cluster-ului la care a fost clasificat fiecare cadru de testare. Chiar dacă antrenarea modelului presupune o abordare nesupervizată, pentru ușurarea evaluării acesteia, setăm numărul de clustere manual. Având aceste date și folosind etichetele video-urilor de testare am putut determina entropia condiționată a rezultatului, aceasta reprezentând performanța descoperirii modelului.

Entropia condiționată se află pe baza a două entități X și Y. X reprezintă etichetele inițiale ale datelor de intrare, care se află în fișierul ce conține video-urile de testare, iar Y reprezintă eticheta sau indexul cluster-ului acestor date în urma aplicării K-means-ului. Entropia condiționată arată de câtă informație am avut nevoie din X ca să aflăm Y. Astfel cu cât este mai mică cu atât performanța modelului este mai mare. Entropia condiționată a modelului antrenat din articol este 3.91, iar pentru modelul antrenat de mine este 2.60. Comparăția nu e neaparat relevantă astfel încât mulțimea de date folosită de mine e de 10 ori mai mică decât cea folosită de autorii articolului original.

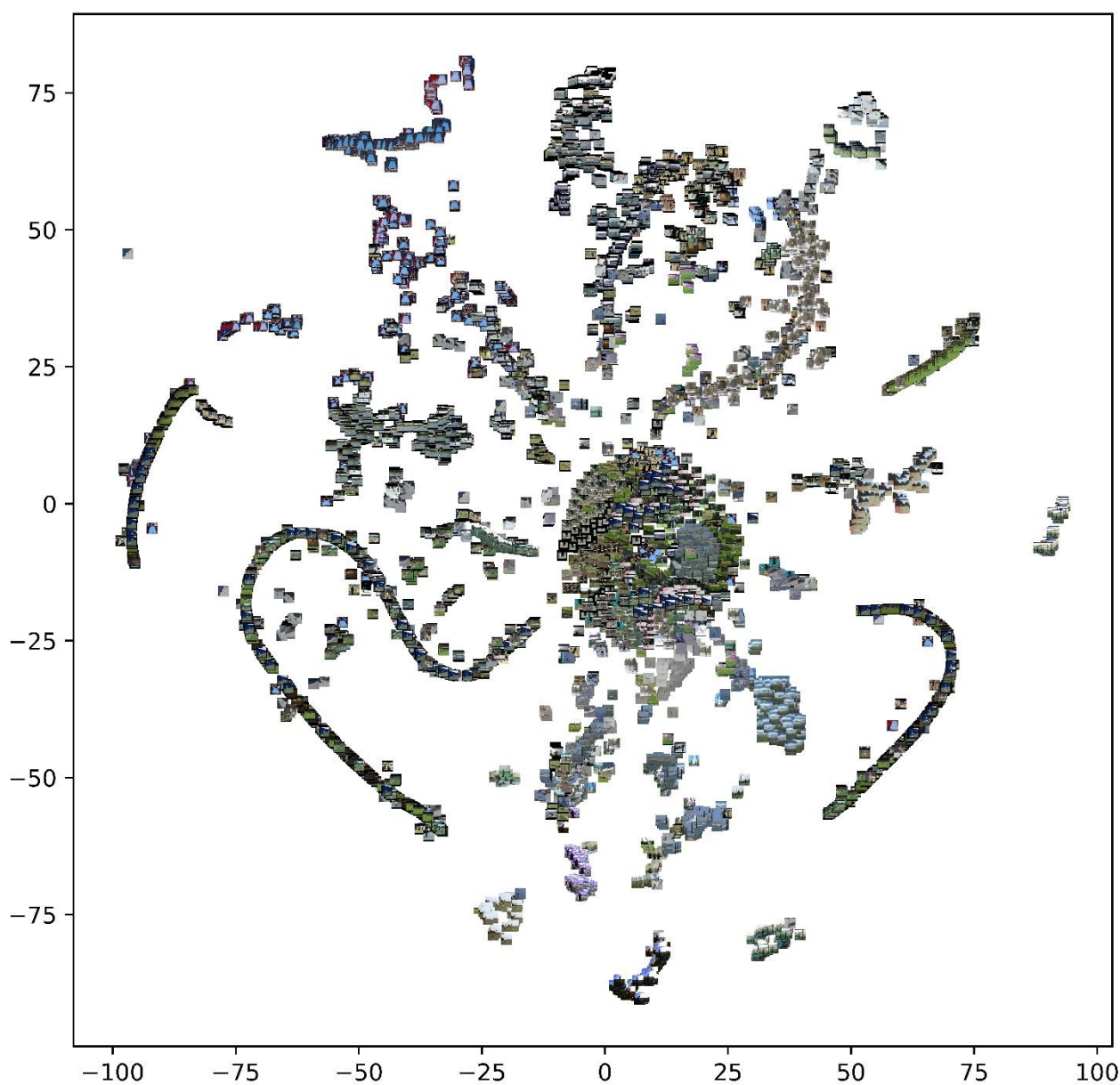
Pentru o vizualizare mai bună a rezultatului, am creat o reprezentare grafică a rezultatului. Din lista cu trăsături a fiecarui cadru am extras reprezentarea acestuia în planul euclidian folosind algoritmul Barnes-Hut tSNE (*t-distributed Stochastic Neighbor Embedding*) din aceeași bibliotecă dar din modulul `manifold`. Astfel o listă de dimensiunea 4096 o reducem la o listă de dimensiunea 2. Apoi având coordonatele cadrelor, am inserat fiecare cadru în plan folosind biblioteca `matplotlib`. Am inserat rezultatele în două imagini.

În **Figura 5** am reprezentat clasificarea cadrelor în urma aplicării algoritmului K-means. Fiecare culoare reprezintă un cluster. Având multe date de intrare reprezentarea lor trebuie făcută pe o imagine cât mai densă pentru a putea fi observate bine punctele diverse. Astfel imaginea a fost creată relativ la unitatea de 600dpi (*dots per inch*).



**Figura 5.** Clasificarea algoritmului K-means a cadrelor din video-urile de testare

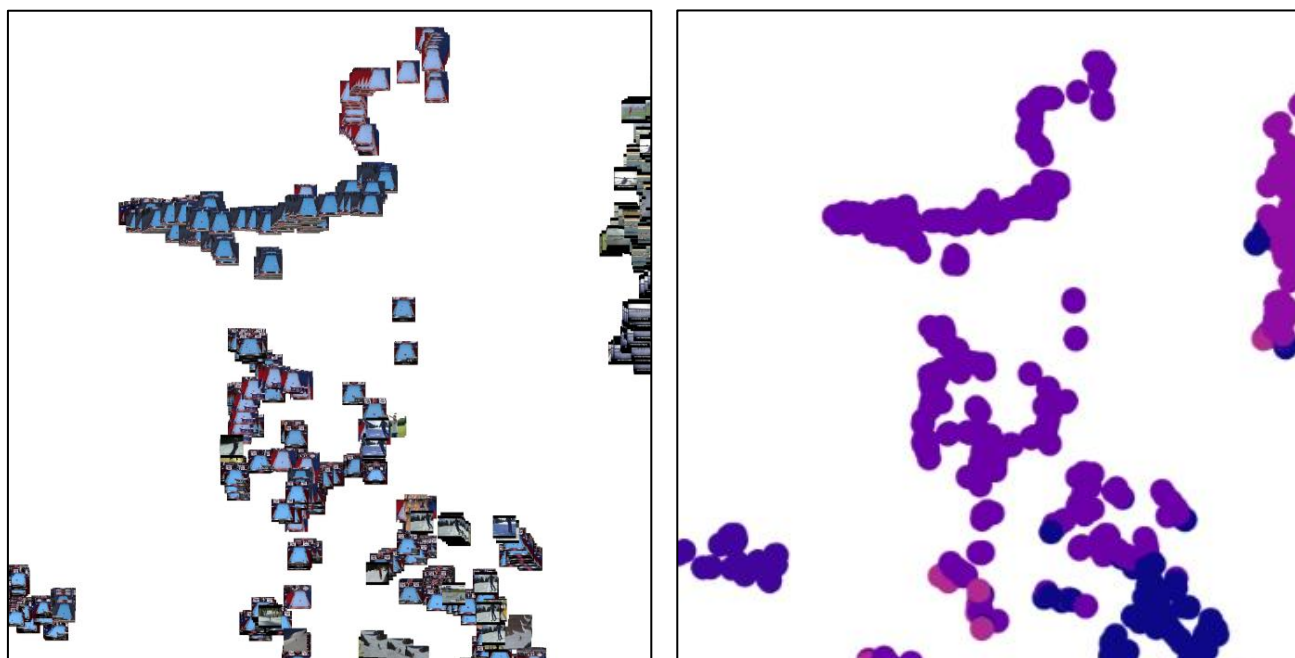
În **Figura 6** este prezentată reprezentarea cadrelor în sine în plan euclidian. Aici se poate vizualiza cel mai bine rezultatele antrenării modelului. Pentru o vizualizare cât mai bună a cadrelor este recomandat folosirea funcției zoom pe documentul digital. A fost necesară mărirea unității de măsurare dpi la 1000 în comparație cu imaginea precedentă, pentru că avem nevoie de mai multă rezoluție pentru vizualizarea mai clară a cadrelor.



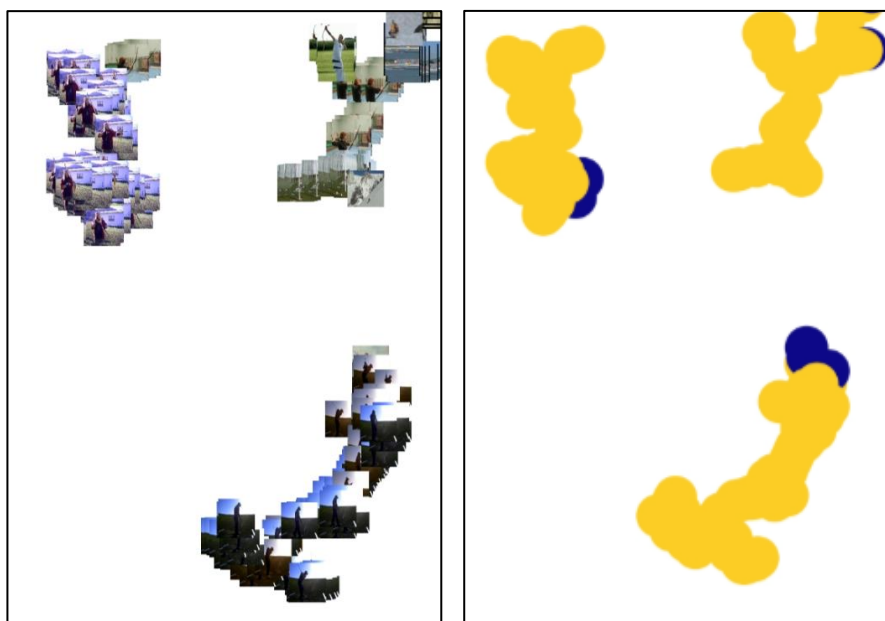
**Figura 6.** Reprezentarea în plan euclidian a trăsăturilor fiecărui cadru procesat.  
Cel mai bine vizualizat cu zoom



În **Figura 7** este prezentat un cluster relativ bine definit după antrenarea modelului și aplicării algoritmului K-means. Se poate observa că marea majoritate a cadrelor din cluster aparțin aceleiași categorii și anume Biliard. În **Figura 8** este prezentat un cluster ce conține cadre similare dar din categorii diferite. Și anume categoriile: tragerea cu arcul, lovitură de golf și aruncarea ciocanului. Toate acestea având în comun talia umană făcând o acțiune.



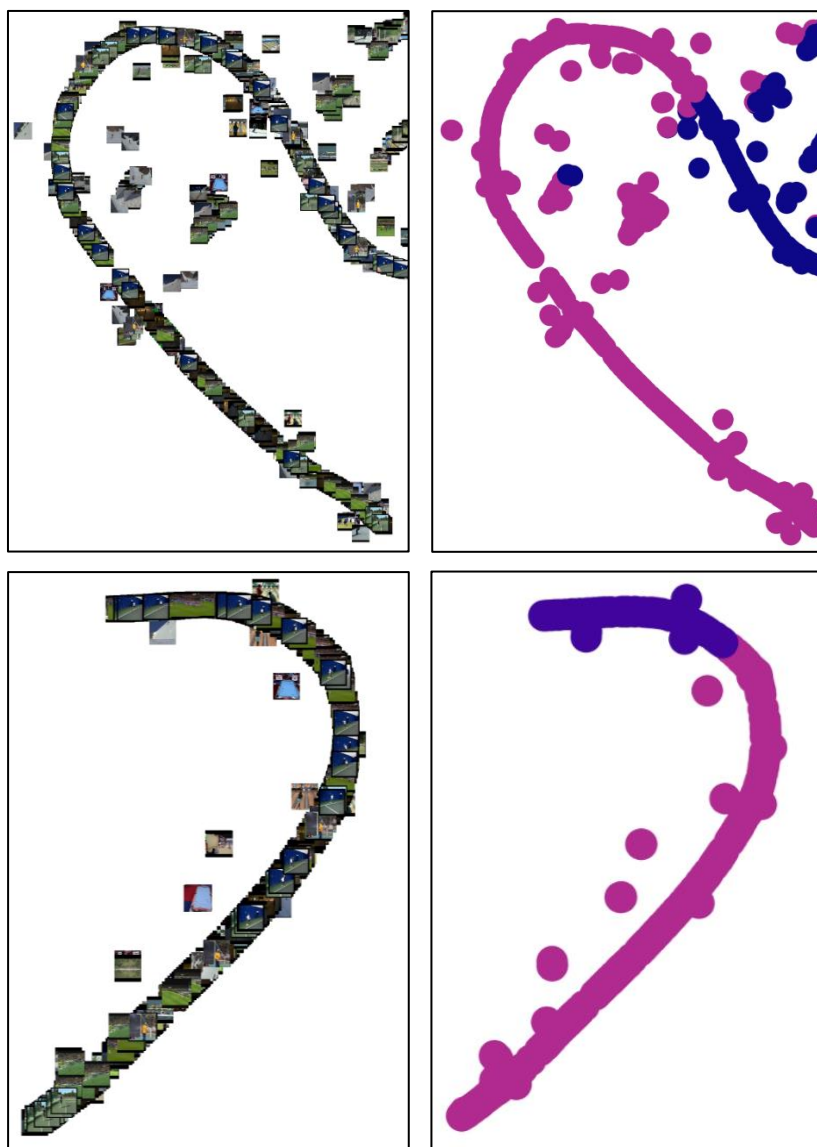
**Figura 7.** Reprezentarea unui cluster bine definit. În partea stângă reprezentarea cadrelor. În partea dreaptă reprezentarea acestor cadre într-un cluster după algoritmul K-means.



**Figura 8.** Reprezentarea unui cluster mai puțin bine definit. În partea stângă reprezentarea cadrelor. În partea dreaptă reprezentarea acestor cadre într-un cluster după algoritmul K-means.

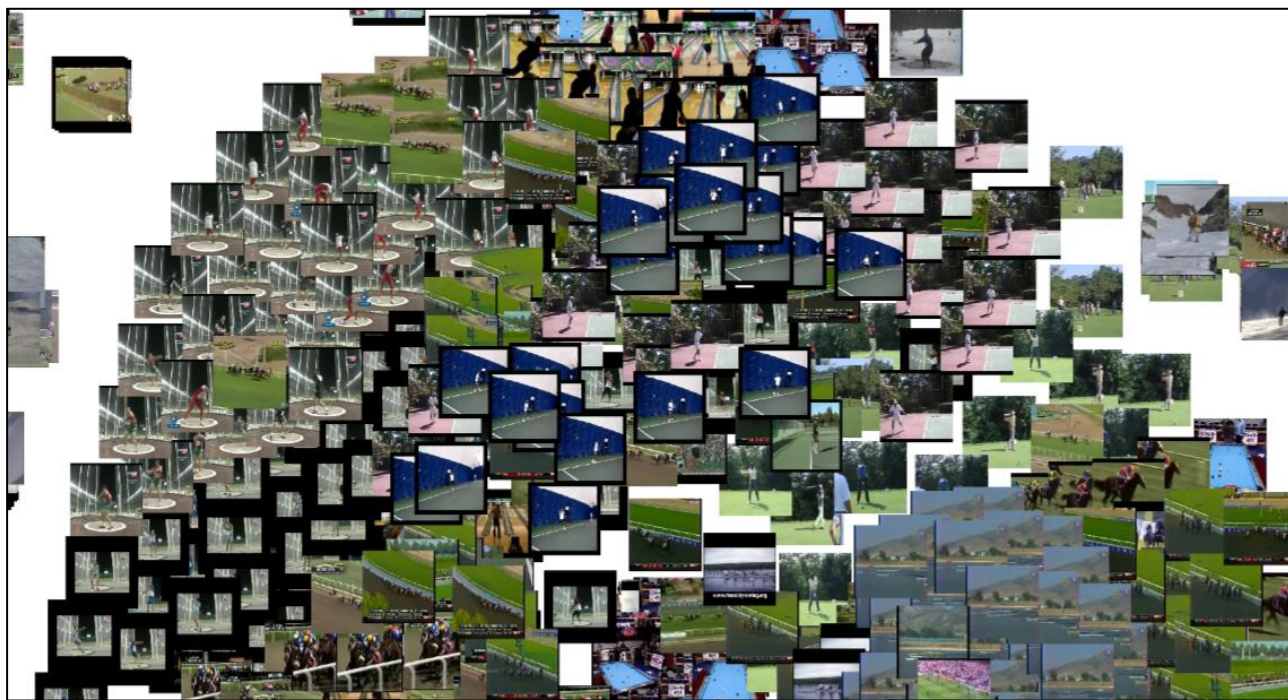


Se poate foarte bine observa și unde modelul a suferit de *overfitting*. Un exemplu din acesta este prezentat în **Figura 9**. În reprezentarea cadrelor se poate observa mai proeminent cadrele ce aparțin atât video-urilor din categoria lovitură de pedeapsă în fotbal cât și din categoria de lovitură de tenis, acestea având unele caracteristici comune, ca de exemplu terenul asemănător ce reprezintă fundalul cadrelor.



**Figura 9:** Exemplu de *overfitting*. În același cluster au fost clasificate cadre din mai multe categorii.

În reprezentare se poate de asemenea observa cum modelul a reușit să învețe să apropie reprezentările trăsăturilor cadrelor din aceleași video-uri în plan euclidian, însă nu a reușit să mărească distanța dintre categorii, sau din video-uri diferite. Exemple în care se întâmplă acest lucru pot fi vizualizate în **Figura 10**. Acest lucru este rezultatul direct al timpului limitat de antrenare a modelului. Deși modelul a fost antrenat doar pentru 500 de iterații, reprezentarea cadrelor inspiră încredere că având mai mult timp, modelul ar fi învățat mai bine.



**Figura 10.** Exemplu de apropiere a trăsăturilor a cadrelor din același video, dar eșuarea în depărtarea reprezentărilor din video-uri diferite

## Concluzii

Prin prezenta lucrare am tratat problema clasificării nesupervizate a video-urilor folosind rețele neuronale, ideea fiind inițial dezvoltată de Redondo-Cabrera et. al. în articolul “*Unsupervised learning from videos using temporal coherency deep networks*”. Având resurse computaționale limitate, am recurs la soluții cloud, aducând astfel un nivel nou de complexitate.

Rezultatele lucrării prezintă doar un mod de executare a antrenării modelului. Orice modificare a anumitor pași intermediari ar putea aduce rezultate diferite.

Având mai mult timp pe mașina virtuală, modelul ar fi învățat mai bine. Astfel am fi putut avea distanțe mai mici între cadrele din aceleași categorii, și distanțe mai mare între cadrele video-urilor din categorii diferite. Eventual am fi putut avea o distingere clară între categorii. Această extindere a antrenării modelului este posibilă deoarece modelul antrenat a fost salvat.

O extindere a acestui experiment, precum o prezintă și autorii lucrării originale, ar fi să folosim ponderile modelului antrenat, și să continuăm antrenarea dar deja supervizată. Astfel începând antrenarea cu un model deja parțial antrenat. Rezultatele acestei extinderi se pot vizualiza în articolul original.

## Bibliografie

[1] 160 YouTube Statistics and Facts (2019) | By the Numbers

<https://expandedramblings.com/index.php/youtube-statistics/>

[2] Unsupervised learning from videos using temporal coherency deep networks

<https://www.sciencedirect.com/science/article/pii/S1077314218301772>

[3] Convolutional Neural Networks

<https://www.coursera.org/learn/convolutional-neural-networks>

[4] Keras

<https://keras.io/>

[5] Keras: A quick overview

<https://www.tensorflow.org/beta/guide/keras/overview>

[6] Unsupervised learning from videos using temporal coherency deep networks: GitHub repository

<https://github.com/gramuah/unsupervised>

[7] Convolution

<https://en.wikipedia.org/wiki/Convolution>

[8] ImageNet Classification with Deep Convolutional Neural Networks

<https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>

[9] Dimensionality Reduction by Learning an Invariant Mapping

<http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf>

[10] UCF101 - Action Recognition Data Set

<https://www.crcv.ucf.edu/data/UCF101.php>

[11] OpenCV

<https://opencv.org/>

[12] Google Cloud Platform

<https://cloud.google.com/>

[13] Python 3.6.5

<https://www.python.org/downloads/release/python-365/>

[14] PyCharm

<https://www.jetbrains.com/pycharm/download>

[15] Tensorflow GPU Support

<https://www.tensorflow.org/install/gpu>

[16] GitHub Issue comment

<https://github.com/tensorflow/tensorflow/issues/22794#issuecomment-442039709>