“Development of an Efficient Deep Learning Model for Sporting Activity Classification”

A PROJECT REPORT

Submitted by

Rishav Giri, Roll No.-1651192, Reg.No-161260110136

Sarasij Jana, Roll No. -1651190, Reg.No-161260110146

Shouvik Chatterjee, Roll No.-1651151, Reg.No-161260110155

Sourav Dey, Roll No.-1651152, Reg.No-161260110172

Under the Supervision of

Prof. Jhalak Dutta (Asst. Professor, Dept. of Computer Science and Engineering)

In partial fulfillment of the requirements for the award of the degree

Of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

HERITAGE INSTITUTE OF TECHNOLOGY, KOLKATA

WEST BENGAL UNIVERSITY OF TECHNOLOGY, KOLKATA

JULY, 2020

HERITAGE INSTITUTE OF TECHNOLOGY, KOLKATA

WEST BENGAL UNIVERSITY OF TECHNOLOGY

BONAFIDE CERTIFICATE

Certified that this project report "**Development of an Efficient Deep Learning Model for Sporting Activity Classification** "is

the bonafide work **of " Rishav Giri, Sarasij Jana, Shouvik Chatterjee and Sourav Dey "**

who carried out the project work under my supervision.

SIGNATURE	**SIGNATURE**
Prof. (Dr.) Subhashis Majumder	Prof. Jhalak Dutta
HEAD OF THE DEPARTMENT	**PROJECT GUIDE**
Department of Computer Science & Engineering,	Department of Computer Science & Engineering,
Heritage Institute of Technology	Heritage Institute of Technology,
P.O. Anandapur, East Kolkata Twp, Kolkata, West Bengal 700107	P.O. Anandapur , East Kolkata Twp, Kolkata, West Bengal 700107

SIGNATURE

EXAMINER

ACKNOWLEDGEMENT

We would take this opportunity to thank Prof. Pranay Chaudhuri, Principal, Heritage Institute of Technology, for providing us with all the necessary facilities to make our project work successful.

We would like to thank our Head of the Department Prof. (Dr.) Subhashis Majumder for his kind assistance as and when required.

We will be thankful to Prof. Jhalak Dutta, our project coordinator for constantly supporting and guiding us and for giving us invaluable insights. His guidance and his words of encouragement motivated us to achieve our goal and provided an impetus to excel.

We thank our faculty members and laboratory assistants at the Heritage Institute of Technology for playing a pivotal and decisive role during the development of the project. Last but not the least we thank all friends for their cooperation and encouragement that they have bestowed on us.

Rishav Giri

Rishav Giri

Shouvik Chatterjee

Shouvik Chatterjee

Sourav Dey

Sourav Dey

Sarasij Jana

Sarasij Jana

TABLE OF CONTENTS

TABLE OF CONTENTS

1. Abstract

This project aims to apply deep neural networks to classify video clips in applications used to streamline advertisements on the web. The system focuses on sport clips but can be expanded into other advertisement fields with lower accuracy and longer training times as a consequence.

2. Introduction

Since the beginning of the decade deep neural networks have established them- selves as the most popular machine learning technology. They are currently being applied to a wide selection of fields including disease detection, automatic tagging in Facebook and forecasting weather and market prices. The deep structures are applicable to almost any kind of problem which has generated a large increase in research and work in the area. Image classification tasks has long been the most popular branch of problems that neural networks can solve. The last few years, giants such as Microsoft and Google have entered the prominent machine learning competition ILSVRC, where they won by using deep neural networks to classify images belonging to a wide range of categories.

This thesis project will approach the field of video classification with the help of deep neural networks. An outline of the report is presented below with a description of each of the main sections. Each section has been designed to be as independent as possible to make them understandable and interesting from a standalone perspective.

3. Problem Description

The project involves the computational fields of video classification and deep neural networks. The main idea is to implement a system capable of classifying the frames in a video clip into different categories. These categories are based on an advertisement taxonomy used as an industry standard. The goal is to be able to take a video clip and correctly classify it's 'activity' into the right category with a reasonable correct percentage. To narrow down the field, we will only focus on sport clips in this project. The classification task then comes down to determining which sport is present in the video. The system should however be straightforward to expand into other areas of the taxonomy, at the possible expense of accuracy and training time. Our main aim is to use a suitable neural network model to classify sports videos.

4. Potential Application Areas

There are a number of possible application areas for the system described above. One possible scenario is the screening of a website to determine its content. The video classification system could be used to identify what kind of videos are avail- able on the website and then use this information to achieve various purposes. In general, the main use of the system is to streamline advertisements on the web to better suit the user.

Another application area is when a person is surfing on the web looking on video clips. A video classification software could then be used to identify what the user is looking at while connecting it to a certain category in the advertisement taxonomy, let's take football as an example. This information could then be used to determine which kind of advertisement to show to the user while he/she is browsing in the future. If the user watches a lot of football clips, the system could recognize this and apply the knowledge to present the user with football advertisements.

The main idea is to increase online sales by turning videos into online stores. By identifying which kind of objects that are present in a video or by linking the video to one of the taxonomy categories, the user could be presented with commercials or products directly related to it.

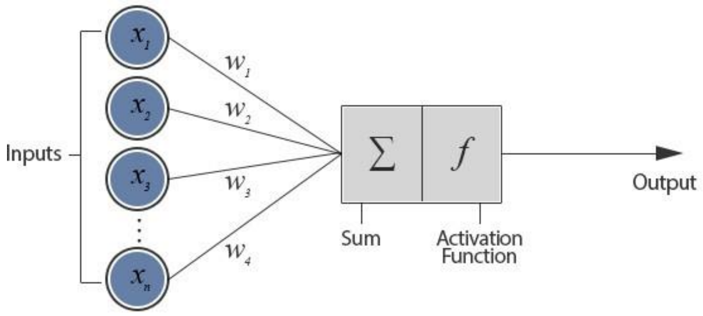
4.1 Novelty of our work

- Our classifier requires only about 600Mb of data to obtain peak accuracy of 92%. Most standard activity recognizers function at an average of 80-85% accuracy.
- The total uncompressed source code size is around 10Mb.It can easily be incorporated in a Flask Application.
- Our system is computationally cheaper than any LSTM/RNN approach and can be independent/used in conjunction with a compatible LSTM.
- The dataset it is trained on is relatively small and self-curated, with augmented images, thereby ensuring minimum assembly and training overheads.

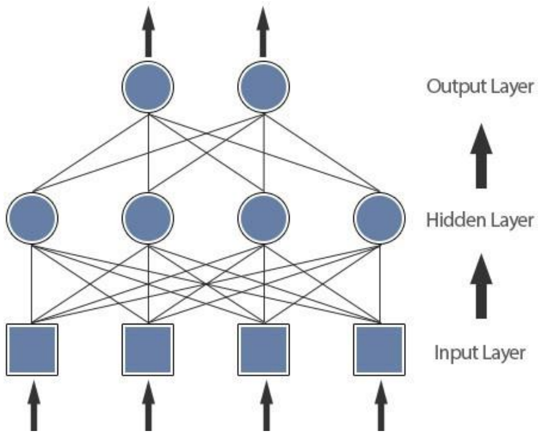
5. Background

Artificial neural networks

Artificial neural networks, simply referred to as neural networks or NNs, were constructed to solve a variety of different problems. These include pattern recognition, classification tasks and forecasting events. The artificial neural network mimics the structure of biological neural network on a basic level. While a nerve cell is made up of a soma, dendrites and an axon, artificial neurons consist of a processing unit, input connections and an output connection.



An artificial neuron a.k.a. a perceptron. Each input has its own corresponding weight. The cell body, or the processing unit, is where the input weights are summarized and then passed into the activation function. There are several types of activation functions. Once the processing is done the artificial neuron will produce an output



A feedforward neural network with one hidden layer. The name feedforward refers to the uni-directed connections between the nodes. The network also satisfies the rules of a multilayer perceptron

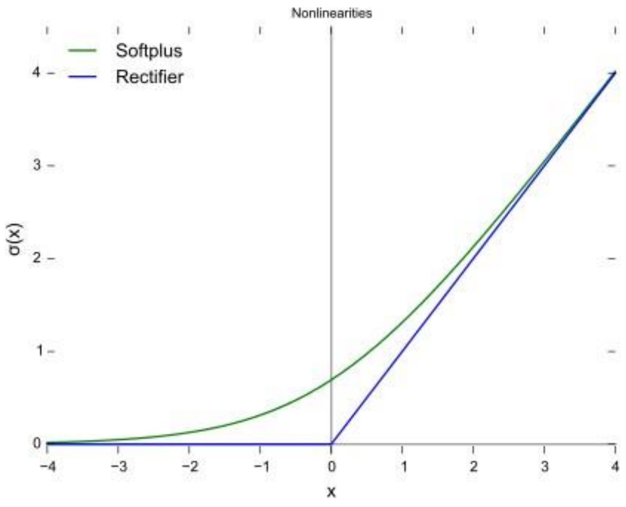
For deep neural networks, the most popular activation function is the rectifier linear unit or the ReLU. It can be described with the formula:

f (x) = max(0, x)

An approximation of the ReLU function called softplus was created to get rid of the hard saturation at 0:

f (x) = log(1 + ex)

However the research does not prove that this smooth approximation enhances the performance.



A plot showing the ReLU activation function and its smooth approximation called softplus. ReLU is the most popular activation function today since it is computationally cheap, biologically plausible and helps the network keep sparsity in its representations

Training neural networks

The main idea of neural networks is that they can learn from training. By training a NN, the weights of the network are updated. This means that we do not have to hard code the values of the network, which would take an eternity to achieve with large networks. There are two main approaches to training an artificial NN: Supervised learning and unsupervised learning.

Supervised learning

Supervised learning means that both the data and the desired result are pro- vided during training. We have our data X and we know what category Y it belongs to.

Unsupervised learning

Unsupervised learning trains the network with a data set X, but do not know which category Y the training data belongs to. An example of use for unsupervised learning is clustering problems.

Perceptron training and the delta rule

The main idea is to compare the desired output to the output generated by the input. Since we have a desired output, the algorithm below belongs in the supervised learning category. We define the following parameters:

- x_i, $i = 1, ..., n$ is the value of the i:th input node
- y is the output value
- d is the desired output value
- t is the current training step
- w_i is the weight of connection between the i:th input node and the output node.
- η is the learning rate.
- g is the activation function.
- h is the weighted sum of all input values $x_1, ..., x_n$ (before applying the activation function).

When we want to update the weight w_i, we can simply use the formula:

$\Delta w_i = \eta(d-y)x_i g'(h)$. $w_i(t+1) = w_i(t) + \Delta w_{ij}$.

Overfitting and regularization

This refers to the problem that occurs when a model more complex than the task requires is used. One example is having too many neurons or layers in the network relative to the amount of training data. Overfitting may lead to worse performance and less generality. It is not to be confused with overtraining which simply means training the network for too long time.

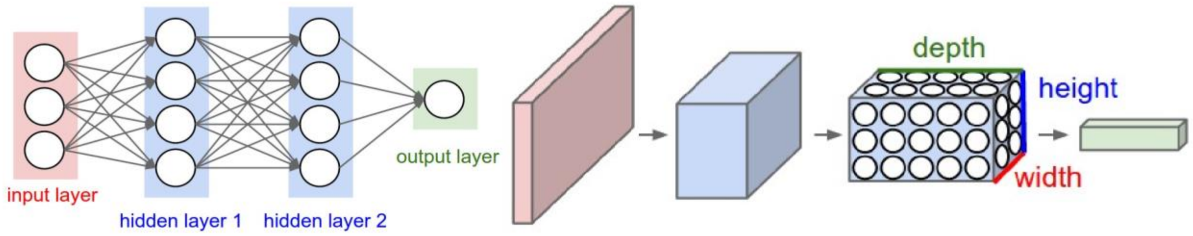
To prevent overfitting we can apply something called regularization. Regularization solves overfitting by adding additional information to the network.

2

Convolutional neural networks (CNN)

We are now discussing the neural network architecture that will be used in the implementation part of this project. Convolutional neural networks have been proved to be very efficient when it comes to classification problems, which is the area we are dealing with in this report. It is a powerful model for both image and video recognition. CNNs have become increasingly popular the last few years and is the basis of many modern computer vision applications, such as self-driving cars or face recognition in Facebook's auto-tag technology.

Even though CNNs have gained much popularity the last few years, it is not a new architecture. It was first proposed in 1989 by Yann LeCun, known as the founding father of convolutional nets. In the article ”Backpropagation applied to zip code recognition” by LeCun et al. a new NN architecture was presented to deal with image recognition tasks. Unlike previous attempts at the same type of problems, the network was now fed directly with raw images instead of vectors containing features from the images. To get the features of the images, convolutional maps were used to extract this information in the early layers of the network. They now had a network capable of both feature extraction and classification that still worked with backpropagation training; the convolutional neural network, or the CNN . This kind of structure was later expanded on by LeCun and in 1998 he and a group of researchers had created a CNN named LeNet-5 which looks much like modern CNNs on many levels. But what exactly is a CNN? By making a comparison with a regular NN the difference becomes clear.



Say that we want a feedforward NN with hidden layers to be able to classify images that have the size 32x32x3 (32x32 pixels and the 3 color channels, RGB). If we want the network to be able to handle all of the picture it means that the input layer has to consist of 32x32x3 = 3072 nodes. This means that the neurons in the hidden layer all have 3072 weighted connections to the input layer, assuming that the layer is fully connected. This is a large number to work with and then we have to consider that a 32x32 pixels image is not particularly large. If we want to process even bigger images, the structure described above becomes inefficient and prone to overfitting. So we either have to downscale the images or try to extract important features before sending them into the regular NN. A CNN does both of these things for us.

The CNN is built to improve performance for image inputs. One big difference is that the CNN often organizes the neurons in three dimensions unlike the regular network which only uses two dimensions. In this 3D environment, the layers are not fully connected to the layer before. Instead we use convolutional maps over the input image to extract features and then compute the output. This means that a layer is only connected to a local area of the previous layer. This "local receptive field" is one of the signature architectural ideas that lies behind the CNN. Another important aspect of CNNs is the "weight sharing". If one of the feature maps are usable in one part of the image it is likely to be useful across the entire image, which is why some parameters can be shared across different convolutions. CNNs can be divided into different types of layers, each with its own specific task. The layer that handles the feature extraction with the help of convolutional maps is simply called the convolutional layer. Other than that we have the input layer, the ReLU layer, the Pooling layer and the fully-connected layer. Below these layers are explained, assuming we want our network to classify an image.

The input layer

The input layer of a CNN works like we are used to, except it usually comes in three dimensions. The width and height of the layer represents the vertical and horizontal pixels of the image while the depth represents the RGB color channels.

The convolutional layer

In this layer the network tries to find features in the image. As can be heard in the name, this layer is where the convolution occurs. The layer uses several filters which are passed over the image to try to find important details that can be used for classification. We can imagine the process as a looking glass sweeping over the image, trying to identify important traits such as edges. The result of this action is a couple of so called activation maps containing the extracted features. The size of these activation maps will be based on the amount of steps it takes for a filter to cover the whole image. The amount of steps required will depend on the size of the input, the filter size and the stride, which is basically how much the filter is shifted at each step. The number of activation maps will be the same as the number of filters used. Handling filters at the edges of the images can be done by padding. With padding, all the elements within the reach of the filter that falls outside of the target image are taken to be a desired value, for example zero.

Using several convolutional layers in a CNN means that very high level features can be extracted. In the first layer, the network could detect simple edges and then in the next layer those edges could be filtered into simple shapes and so on.

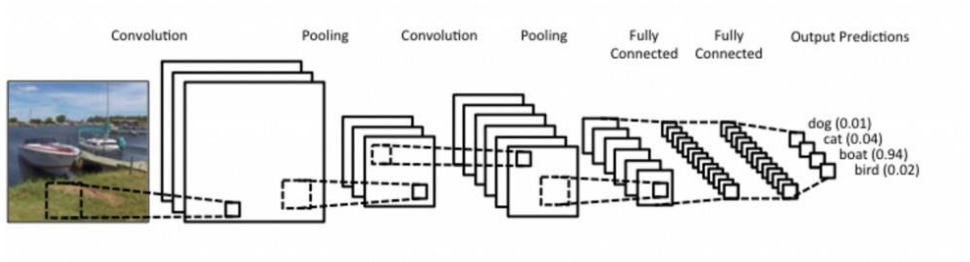


Illustration of a CNN. The picture shows the network from the feature ex- traction and downsampling in the beginning to the classification part in the end. Left to right, we first have the input image where one of the filters used in the first convolutional layer can be seen in the lower part of the image. Then in the next part we can see the resulting activation maps. Three activation maps means that three filters were used.

(Continued) In the following step we see the activation maps being compressed through downsampling in the pooling layer. Then the convolution and the pooling are repeated once and at last we have our fully connected layers handling the classification

The ReLU layer

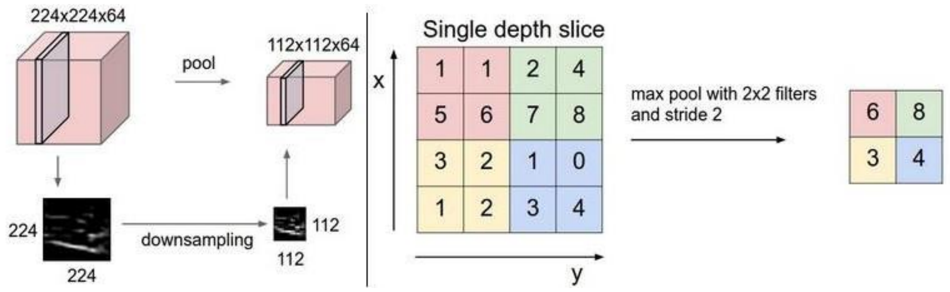
The ReLU layer basically applies the ReLU activation function shown in the Neural Networks section. It helps the network keep sparsity in its representations and is computationally cheap. Theoretically any activation function could be used but ReLU is the most popular.

The pooling layer

This layer is also known as the subsampling or downsampling layer. Pooling layers are usually inserted between convolutional layers. When features have been extracted from images, the locations of these features are not very important.

The thing that remains important is their position in relation to each other. The use of the precise position of the features has actually been shown to be harmful to the performance of neural networks. The position will probably vary for different versions of the same input and therefore training a network to look for a specific feature at a specific spot is not preferred. With subsampling we can both reduce this problem and reduce the amount of parameters and computation needed later in the network. We want to lower the spatial size to make the data easier to handle and to prevent overfitting.

At the same time we do not want to lose too much important information. The most common pooling method is max pooling. Patches are applied to the data, much like in the convolutional layer, but here the max value from each patch is selected. Much like the filters in the convolutional layers, one can alter the stride to determine the step size of the pooling. When working on activation maps, the pooling is done independently on each map. In the next figure, the use of max pooling to downsample an image is illustrated.



Max pooling downsamples the data to optimize the network and to reduce overfitting. Here we can see a 224x224 image being downsampled to 112x112 pixels through max pooling with filter size 2x2 and stride 2

The fully-connected layer

This is the only part of the CNN that is fully connected to the previous layer. It is the part of the CNN that classifies the feature data extracted and down- sampled in previous layers. Its structure is very similar to an ordinary MLP. By now the data is smaller thanks to the pooling layers and the computations are easier to handle. The fully connected layers are (usually) the last part of the CNN. At the very end we find a loss function, most commonly the Softmax function. This is similar to regular NNs used for classification.

Formulae:

1. A feed-forward neural network can be thought of as the composition of number of functions

$$f(x)=f_L(\ldots f_2(f_1(x;w_1);w_2)\ldots),w_L).f(x)=f_L(\ldots f_2(f_1(x;w_1);w_2)\ldots),w_L).$$

Each function $f_l f_l$ takes as input a datum $x_l x_l$ and a parameter vector $w_l w_l$ and produces as output a datum $x_{l+1} x_{l+1}$. While the type and sequence of functions is usually handcrafted, the parameters $w=(w_1,\ldots,w_L)w=(w_1,\ldots,w_L)$ are *learned from data* in order to solve a target problem, for example classifying images or sounds.

2. The simplest non-linearity is obtained by following a linear filter by a *non-linear activation function*, applied identically to each component (i.e. point-wise) of a feature map. The simplest such function is the *Rectified Linear Unit (ReLU)*

$$y_{ijk}=\max\{0,x_{ijk}\}.$$

3. Another important CNN building block is channel-wise normalisation. This operator normalises the vector of feature channels at each spatial location in the input map xx. The form of the normalisation operator is actually rather curious:

$$y_{ijk'}=x_{ijk}(\kappa+\alpha\sum k\in G(k')x^2{}_{ijk})\beta y_{ijk'}=x_{ijk}(\kappa+\alpha\sum k\in G(k')x_{ijk}{}^2)\beta$$

where $G(k)=[k-\lfloor\rho^2\rfloor,k+\lceil\rho^2\rceil]\cap 1,2,\ldots,KG(k)=[k-\lfloor\rho^2\rfloor,k+\lceil\rho^2\rceil]\cap 1,2,\ldots,K$ is a group of $\rho\rho$ consecutive feature channels in the input map.

4. The parameters of a CNN $w=(w_1,\ldots w_L)w=(w_1,\ldots w_L)$ should be learned in such a manner that the overall CNN function $z=f(x;w)z=f(x;w)$ achieves a desired goal. In some cases, the goal is to model the distribution of the data, which leads to a *generative objective*. Here, however, we will use ff as a *regressor* and obtain it by minimising a *discriminative objective*. In simple terms, we are given:

- examples of the desired input-output relations $(x1,z1),\ldots,(xn,zn)(x1,z1),\ldots,(xn,zn)$ where xixi are input data and zizi corresponding output values;
- and a loss $\ell(z,z\hat{})\ell(z,z\hat{})$ that expresses the penalty for predicting $z\hat{}z\hat{}$ instead of zz.

We use those to write the empirical loss of the CNN ff by averaging over the examples:

$$L(w)=1n\sum i=1n\ell(z_i,f(x_i;w))$$

5. Training CNNs is normally done using a gradient-based optimization method. The CNN ff is the composition of LL layers $f_l f_l$ each with parameters $w_l w_l$, which in the simplest case of a chain looks like:

$$x_0 \longrightarrow f_1 \uparrow w_1 \longrightarrow x_1 \longrightarrow f_2 \uparrow w_2 \longrightarrow x_2 \longrightarrow \cdots \longrightarrow x_{L-1} \longrightarrow f_L \uparrow w_L \longrightarrow x_L x_0 \longrightarrow f_1 \uparrow w_1 \longrightarrow x_1 \longrightarrow f_2 \uparrow w\text{-}$$
$$\longrightarrow x_2 \longrightarrow \cdots \longrightarrow x_{L-1} \longrightarrow f_L \uparrow w_L \longrightarrow x_L$$

During learning, the last layer of the network is the *loss function* that should be minimized. Hence, the output $x_L = x_L x_L = x_L$ of the network is a **scalar** quantity (a single number).

6. The gradient is easily computed using using the **chain rule**. If *all* network variables and parameters are scalar, this is given by:

$$\partial f \partial w_l(x_0;w_1,\ldots,w_L) = \partial f L \partial x_{L-1}(x_{L-1};w_L) \times \cdots \times \partial f l + 1 \partial x_l(x_l;w_{l+1}) \times \partial f l \partial w l(x_{l-1};w_l)$$

7. The CNN computes as per our definition and optimization of the following objective function:

$$E(w,b) = \lambda 2 \| w \| 2 + 1 \,|P|\, \sum (u,v) \in P \max\{0, 1 - f(x;w,b)(u,v)\} + 1 \,|N|\, \sum (u,v) \in N \max\{0, f(x;w,b)(u,v)\}.$$

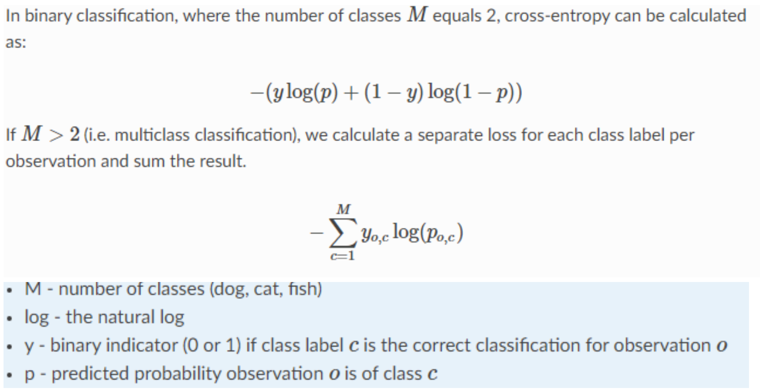
We train the CNN by minimising the objective function with respect to ww and bb. We do so by using an algorithm called *gradient descent with momentum*. Given the current solution $(w_t,b_t)(w_t,b_t)$, this is updated to $(w_{t+1},b_{t+1})(w_{t+1},b_{t+1})$ by following the direction of fastest descent of the objective $E(wt,bt)E(wt,bt)$ as given by the negative gradient $-\nabla E - \nabla E$. However, gradient updates are smoothed by considering a *momentum* term $(\bar{w}\ t, \bar{\mu}\ t)(\bar{w}\ t, \bar{\mu}\ t)$, yielding the update equations

$$\bar{w}_{t+1} \leftarrow \mu \bar{w}\ t + \eta \partial E \partial wt, w_{t+1} \leftarrow w_t - \bar{w}\ t. \bar{w}_{t+1} \leftarrow \mu \bar{w}_{t+\eta} \partial E \partial w_t, w_{t+1} \leftarrow w_t - \bar{w}_t.$$

and similarly for the bias term. Here $\mu\mu$ is the *momentum rate* and $\eta\eta$ the *learning rate*.

Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or sub-differentiable). It can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in high-dimensional optimization problems this reduces the computational burden, achieving faster iterations in trade for a lower convergence rate.

Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value. A perfect model would have a log loss of 0.



In binary classification, where the number of classes M equals 2, cross-entropy can be calculated as:

$$-(y\log(p) + (1-y)\log(1-p))$$

If $M > 2$ (i.e. multiclass classification), we calculate a separate loss for each class label per observation and sum the result.

$$-\sum_{c=1}^{M} y_{o,c}\log(p_{o,c})$$

- M - number of classes (dog, cat, fish)
- log - the natural log
- y - binary indicator (0 or 1) if class label c is the correct classification for observation o
- p - predicted probability observation o is of class c

Action recognition

Action recognition task involves the identification of different actions from video clips (a sequence of 2D frames) where the action may or may not be performed throughout the entire duration of the video. This seems like a natural extension of image classification tasks to multiple frames and then aggregating the predictions from each frame. Despite the stratospheric success of deep learning architectures in image classification (ImageNet), progress in architectures for video classification and representation learning has been slower.

What made this task tough?

Huge Computational Cost -A simple convolution 2D net for classifying 101 classes has just ~5M parameters whereas the same architecture when inflated to a 3D structure results in ~33M parameters.

Capturing long context-Action recognition involves capturing spatiotemporal context across frames. Additionally, the spatial information captured has to be compensated for camera movement.

Designing classification architectures-Designing architectures that can capture spatiotemporal information involve multiple options which are non-trivial and expensive to evaluate.

No standard benchmark-The most popular and benchmark datasets have been UCF101 and Sports1M for a long time. Searching for reasonable architecture on Sports1M can be extremely expensive.

Before deep learning came along, most of the traditional CV algorithm variants for action recognition can be broken down into the following 3 broad steps:

1. Local high-dimensional visual features that describe a region of the video are extracted either densely or at a sparse set of interest points.
2. The extracted features get combined into a fixed-sized video level description. One popular variant to the step is to bag of visual words (derived using hierarchical or k-means clustering) for encoding features at video-level.
3. A classifier, like SVM or RF, is trained on bag of visual words for final prediction

Deep neural networks for video classification

In this section the topic of using deep neural networks for classifying videos has been researched further. With a focus on convolutional neural networks (and LSTMs), a couple of known architectures are presented below.

There have been several attempts at achieving good performance video classification with convolutional networks. For example in the models developed in ”Large-scale Video Classification with Convolutional Networks” by Karpathy et al. and ”Learning End-to-end Video Classification with Rank-Pooling” by Fernando and Gould, CNNs are used to classify videos from different large datasets. Karpathy and and his crew tried several different fusion techniques   and compare them to a single frame approach while Fernando and Gould compare different pooling methods such as rank pooling and max pooling. Both models achieved good results with up to 80-85 % accuracy.

Since image classification is similar to video classification, a network handling videos is working with frames which can basically be seen as images. A network capable of handling single images can also handle the frames of a video clip.

What is Deep Residual Network?

Deep Residual Network is almost similar to the networks which have convolution, pooling, activation and fully-connected layers stacked one over the other. The only construction to the simple network to make it a residual network is the ***identity connection*** between the layers.

6. Technology Stack

Hardware, software, environment and framework specifications have been discussed here.

Proposing a framework

In this section a framework for developing neural networks will be presented and the proposed network model that will be used for the classification task will be demonstrated/utilized.

To simplify the implementation part it was decided that an existing neural network framework should be used instead of building a network from scratch. There are several such platforms available and to find out which one was best suited for this project, a small research was initiated. The research was done by trying out the different libraries, consulting with my supervisors and by reading other people's opinions and research. A publication called" Deep Learning with Theano, Torch, Caffe, TensorFlow, and Deeplearning4J: Which One Is the Best in Speed and Accuracy?" by Kovalev et al. was used to determine which library to use as well.

As time was an important resource in this project, a framework that was simple was preferred over a complicated one that might have had better accuracy or faster classification. Community support, a known runtime and the capacity to support several neural network models was a must.

Environment: Python 3.0

OS: Linux RHEL 6.10

Hardware:

Laptop Model: Dell Alienware 17

- CPU: 2.9GHz Intel Core i9-8950HK (hexa-core, 12MB cache, up to 4.8GHz)
- Graphics: Nvidia GeForce GTX 1080 OC (8GB GDDR5X)
- RAM: 32GB DDR4 (2,666MHz)

Framework and Dependencies:

Keras (v 2.3.0)

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.

We used Keras since it is a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

Cv2 (OpenCV) (4.3.0)

An OpenCL-based GPU interface, OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms.

NumPy (1.19.0)

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Matplotlib (3,2,2)

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy.

Scikit-learn (0.23.1)

It is a free machine learning library for Python. It features various algorithms like support vector machine, random forests, and k-neighbors, and it also supports Python numerical and scientific libraries like NumPy and SciPy

Imutils(0.5.3)

A series of convenience functions to make basic image processing functions such as translation, rotation, resizing, skeletonization, displaying Matplotlib images, sorting contours, detecting edges, and much more easier with OpenCV and both Python 2.7 and Python 3.

TensorFlow (for backend) (v 2.0):

TensorFlow is a framework developed by Google that was released as open- source to support innovation and research. It provides a Python API and works with data flow graphs for numerical computation. It supports CNNs but the tutorials available for it requires extensive knowledge of TensorFlow. To make the work a little bit easier, there is a visualization tool called TensorBoard avail- able. According to the research.it shows good speed but the accuracy drops mystically when the number of layers are increased

Platform:

Google Colab (single 12GB **NVIDIA** Tesla K80 **GPU** that can be used up to 12 hours+free TPU for our configuration.Total storage:1.8 GB**)**

Colab (Colaboratory) is a research tool for machine learning education and research. It's a Jupyter notebook environment that requires no setup to use and allows cloud based computation solutions.

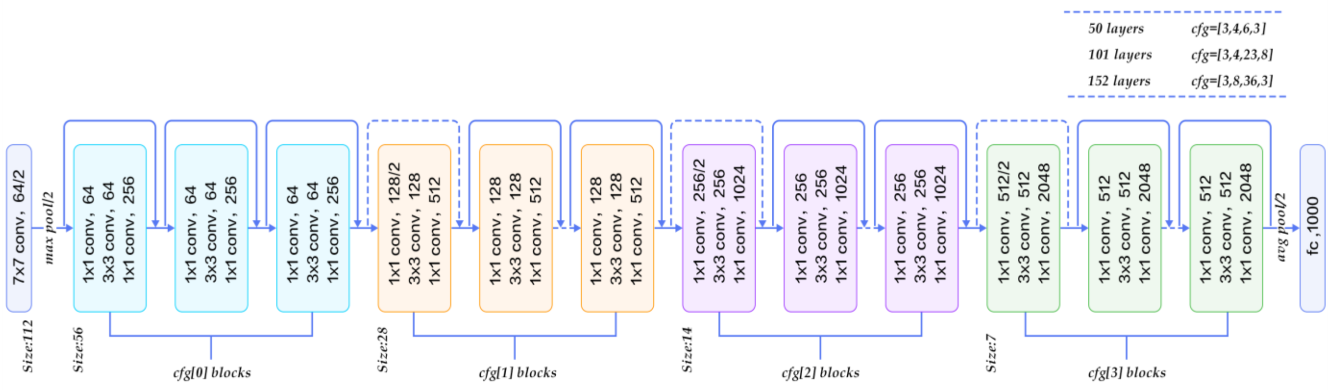
Proposed Neural Network Model:

A CNN network based on ResNet50

The proposed model to be used is a CNN network. It is a robust model and was mainly built to be fast and to give a first look into the classification problem. It contains two convolutional layers, two fully connected layers and one downsampling layer.

Convolutional 1 (Filter size: 14X4, stride: 7, padding: 0, nrOfFilters: 32)
ReLU Activation
Pooling 1 (Filter size: 4X4, stride: 2)
Convolutional 2 (Filter size: 3X3, stride: 2, padding: 0, nrOfFilters: 64)
ReLU Activation
Fully connected (nrOfOutputNodes: 128)
ReLU Activation
Fully connected (nrOfOutputNodes: 64)
ReLU Activation
Softmax Output

6.1 ResNet50 Architecture



ResNet was introduced by **Microsoft, and won the ILSVRC (ImageNet Large Scale Visual Recognition Challenge) in 2015.** It has revolutionized Computer Vision and comes in 3 forms ResNet **50/101/152**.

The **architecture** of **ResNet50** has 4 stages as shown in the diagram. Every **ResNe**t architecture performs the initial convolution and max-pooling using 7×7 and 3×3 kernel sizes respectively.

Key Features of ResNet:

- ResNet uses Batch Normalization at its core. The Batch Normalization adjusts the input layer to increase the performance of the network. The problem of covariate shift is mitigated.
- ResNet makes use of Identity Connection, which helps to protect the network from vanishing gradient problem.
- Deep Residual Network uses bottleneck residual block design to increase the performance of the network.

7. Inherent Problem Identification (Predictive Flickering) and Solution:

Videos can be understood as a series of individual images; and therefore, many deep learning practitioners would be quick to treat video classification as performing image classification a total of *N* times, where *N* is the total number of frames in a video.

There's a problem with that approach though.

Video classification is *more* than just simple image classification — **with video we can typically make the assumption that subsequent frames in a video are *correlated* with respect to their *semantic contents.***

If we are able to take advantage of the temporal nature of videos, we can improve our actual video classification results.

Neural network architectures such as Long Short-Term Memory (LSTMs) and Recurrent Neural Networks (RNNs) are suited for time series data — but in our case, they may be an overkill. They are also resource-hungry and time-consuming when it comes to training over thousands of *video* files.

Instead, for our application, it is sufficient for us to use *rolling averaging* over predictions to reduce "flickering" in results (the problem).

How video classification different than image classification->

When performing image classification, we:

1. Input an image to our CNN
2. Obtain the predictions from the CNN
3. Choose the label with the largest corresponding probability

Since a video is just a series of frames, a naive video classification method would be to:

1. Loop over all frames in the video file
2. For each frame, pass the frame through the CNN
3. Classify each frame *individually* and *independently* of each other
4. Choose the label with the largest corresponding probability
5. Label the frame and write the output frame to disk

There’s a problem with this approach — if we try to apply simple image classification to video classification, we encounter a sort of **“prediction flickering”** . For example, in our demo visualization we saw our CNN shifting between two predictions: *“football”* and the correct label, *“weight_lifting”*. The video was clearly of weightlifting and we would like our entire video to be labeled as such — by preventing the CNN “flickering” between these two labels

A simple, yet elegant solution, we figured, is to utilize a rolling prediction average.

8. Proposed Algorithm

Our optimized algorithm:

1. Loop over all frames in the video file
2. For each frame, pass the frame through the CNN
3. Obtain the predictions from the CNN
4. Maintain a list of the last K predictions
5. Compute the average of the last K predictions and choose the label with the largest corresponding probability
6. Label the frame and write the output frame to disk

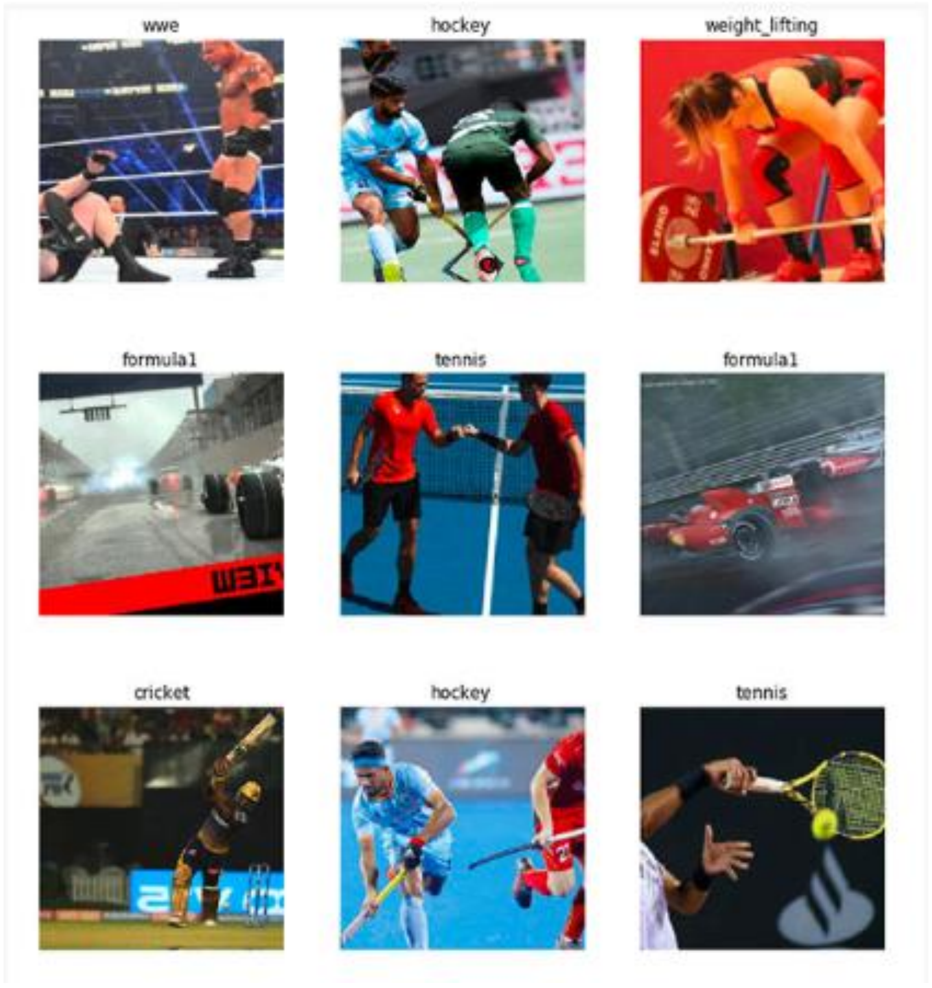
9. Our Sports Classification Dataset

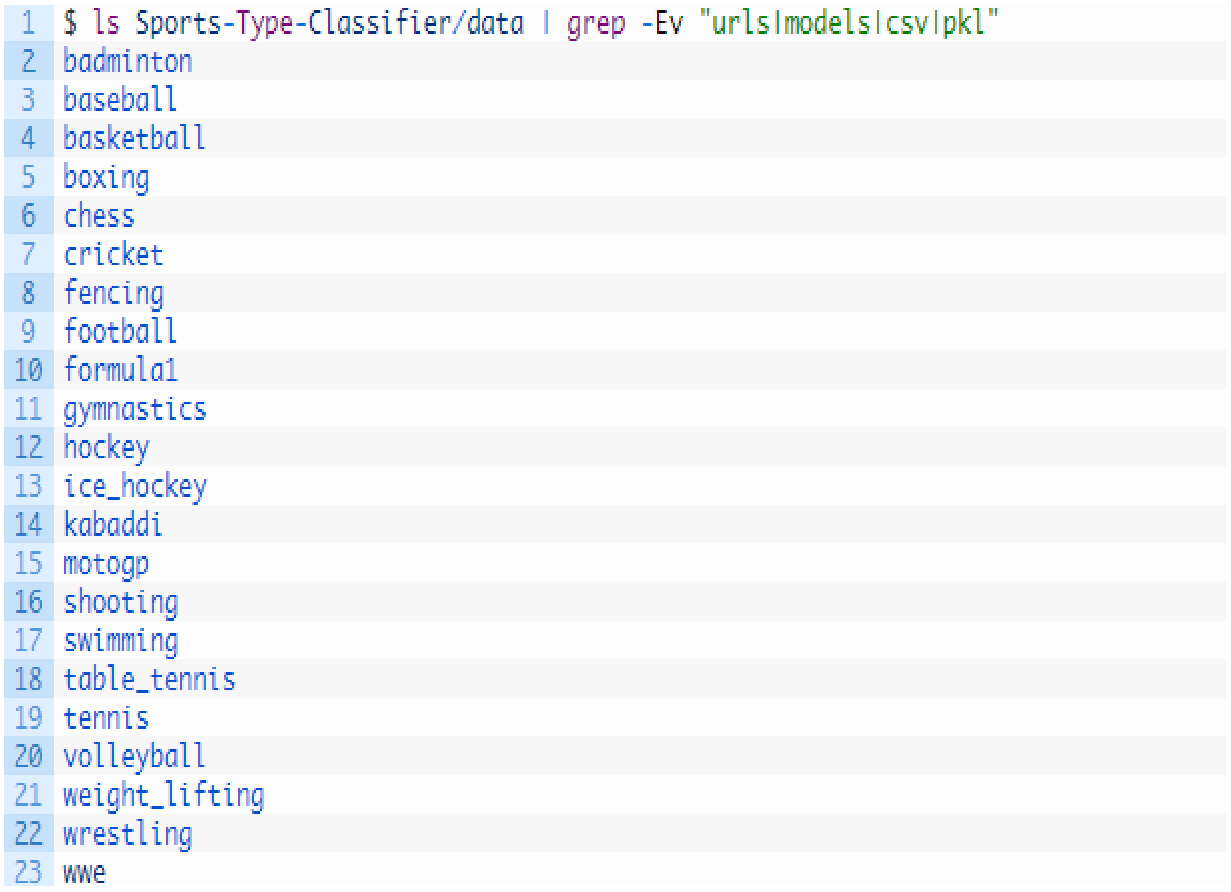
The dataset we be using here is for sport/activity classification. The dataset was curated by downloading photos from Google Images for the following categories:

- Swimming
- Badminton
- Wrestling
- Olympic Shooting
- Cricket
- Football
- Tennis
- Hockey
- Ice Hockey
- Kabaddi
- WWE wrestling
- Gymnasium
- Weight lifting
- Volleyball
- Table tennis
- Baseball
- Formula 1
- Moto GP
- Chess
- Boxing
- Fencing
- Basketball

To save time, computational resources, and to demonstrate the actual video classification algorithm, we trained on a subset of the sports type dataset:

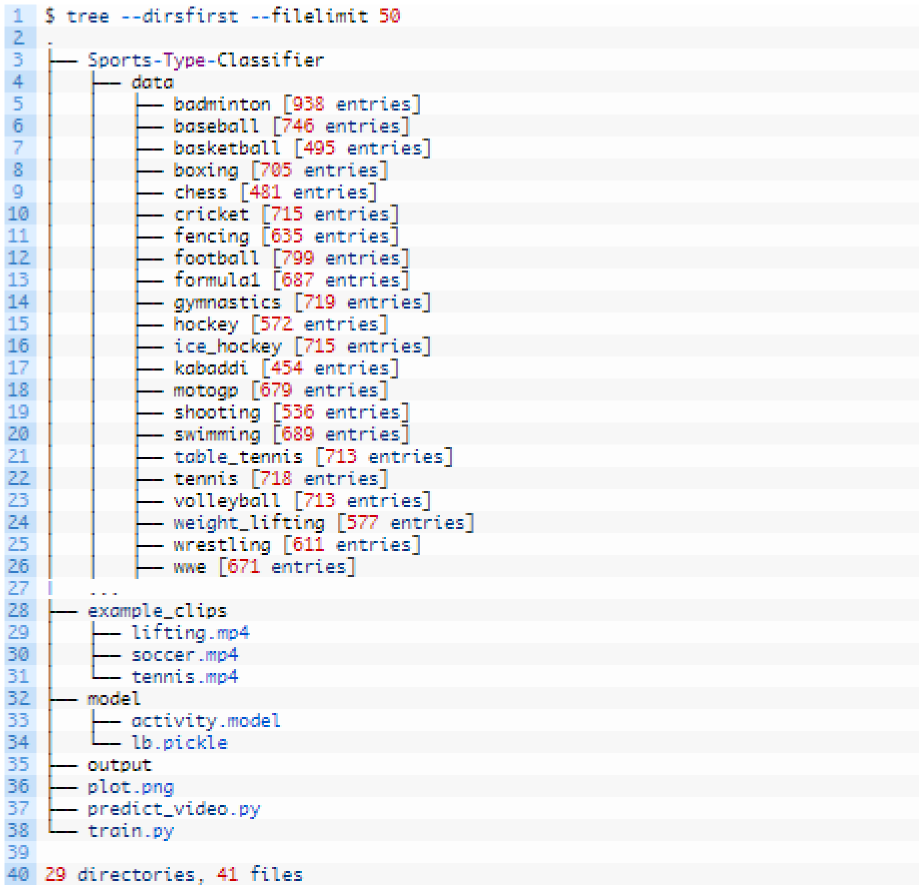
- **Football (i.e., soccer):** 799 images
- **Tennis:** 718 images
- **Weightlifting:** 577 images





```
$ ls Sports-Type-Classifier/data | grep -Ev "urls|models|csv|pkl"
badminton
baseball
basketball
boxing
chess
cricket
fencing
football
formula1
gymnastics
hockey
ice_hockey
kabaddi
motogp
shooting
swimming
table_tennis
tennis
volleyball
weight_lifting
wrestling
wwe
```

Project Structure/Directory Map:



```
$ tree --dirsfirst --filelimit 50
.
├── Sports-Type-Classifier
│   ├── data
│   │   ├── badminton [938 entries]
│   │   ├── baseball [746 entries]
│   │   ├── basketball [495 entries]
│   │   ├── boxing [705 entries]
│   │   ├── chess [481 entries]
│   │   ├── cricket [715 entries]
│   │   ├── fencing [635 entries]
│   │   ├── football [799 entries]
│   │   ├── formula1 [687 entries]
│   │   ├── gymnastics [719 entries]
│   │   ├── hockey [572 entries]
│   │   ├── ice_hockey [715 entries]
│   │   ├── kabaddi [454 entries]
│   │   ├── motogp [679 entries]
│   │   ├── shooting [536 entries]
│   │   ├── swimming [689 entries]
│   │   ├── table_tennis [713 entries]
│   │   ├── tennis [718 entries]
│   │   ├── volleyball [713 entries]
│   │   ├── weight_lifting [577 entries]
│   │   ├── wrestling [611 entries]
│   │   ├── wwe [671 entries]
|   ...
├── example_clips
│   ├── lifting.mp4
│   ├── soccer.mp4
│   └── tennis.mp4
├── model
│   ├── activity.model
│   └── lb.pickle
├── output
├── plot.png
├── predict_video.py
└── train.py

29 directories, 41 files
```

Our training image data is in the Sports-Type-Classifier/data/ directory, organized by class.

Our classifier files are in the model/ directory. Included are activity.model (the trained Keras model) and lb.pickle (our label binarizer).

An empty output/ folder is the location where we store video classification results.

We focused on the following Python scripts:

- train.py : A Keras training script that grabs the dataset class images that we care about, loads the **ResNet50** CNN, and applies transfer learning/fine-tuning of ImageNet weights to train our model. The training script generates/outputs three files:
  - model/activity.model : A fine-tuned classifier based on ResNet50 for recognizing sports.
  - model/lb.pickle : A serialized label binarizer containing our unique class labels.
  - plot.png : The accuracy/loss training history plot.
- predict_video.py : Loads an input video from the example_clips/ and proceeds to classify the video ideally using today's rolling average method.

10. Methodology/Procedural Briefing

FLOWCHART

Dataset population/augmentation

Splitting training and testing data and preprocessing

“Network surgery” as a part of fine-tuning

Customizing the ResNet50 model’s FC Head layer

Compiling and training the model with SGD and categorical cross entropy

Evaluating our network and plotting the training history

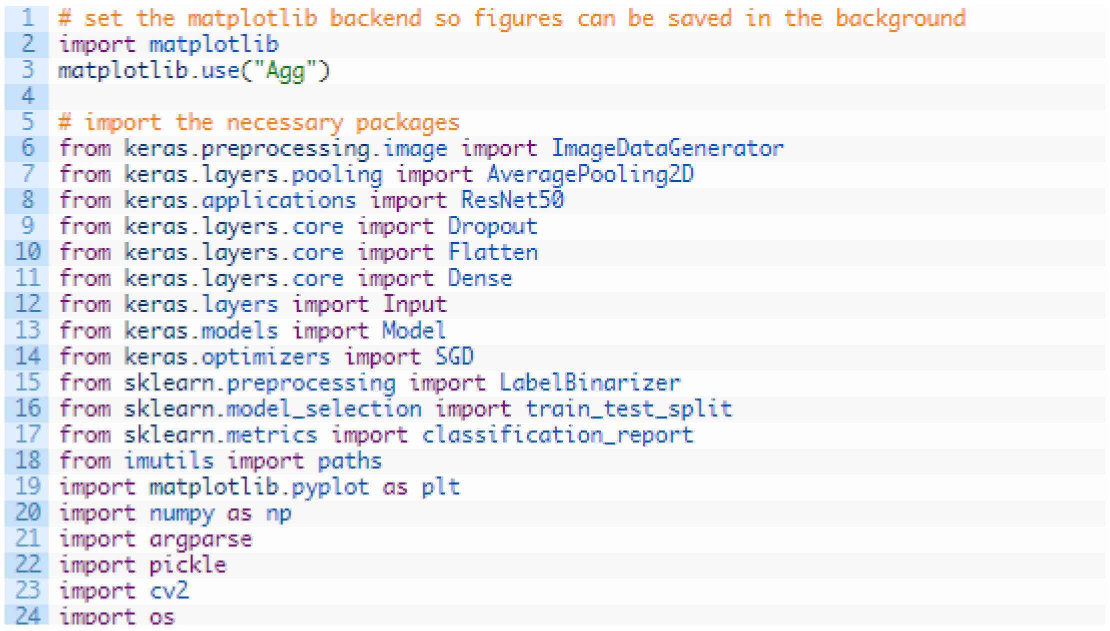
Testing video classification with our rolling prediction algorithm and Cv2 Video frame capturing

Evaluating results and generating performance metrics/classification report

10.1 Training (Overview of source code)

Implementing our classifier training script

Implementing our training script used to train a Keras CNN to recognize each of the sports activities.



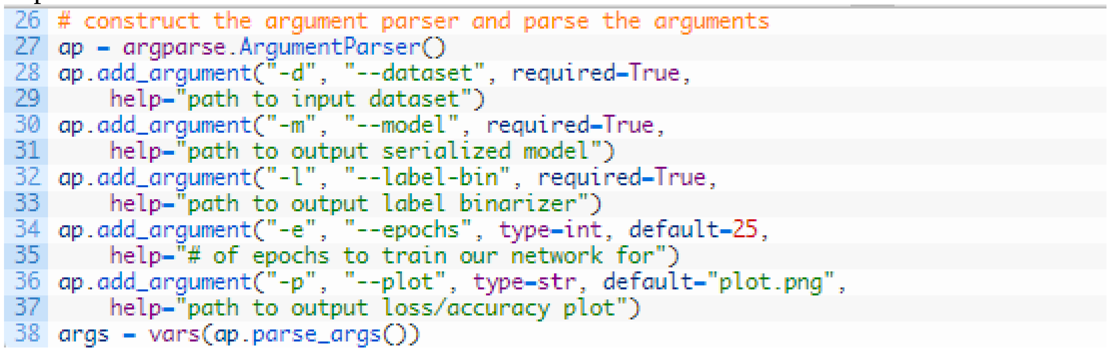
```
 1 # set the matplotlib backend so figures can be saved in the background
 2 import matplotlib
 3 matplotlib.use("Agg")
 4
 5 # import the necessary packages
 6 from keras.preprocessing.image import ImageDataGenerator
 7 from keras.layers.pooling import AveragePooling2D
 8 from keras.applications import ResNet50
 9 from keras.layers.core import Dropout
10 from keras.layers.core import Flatten
11 from keras.layers.core import Dense
12 from keras.layers import Input
13 from keras.models import Model
14 from keras.optimizers import SGD
15 from sklearn.preprocessing import LabelBinarizer
16 from sklearn.model_selection import train_test_split
17 from sklearn.metrics import classification_report
18 from imutils import paths
19 import matplotlib.pyplot as plt
20 import numpy as np
21 import argparse
22 import pickle
23 import cv2
24 import os
```

train.py

On **Lines 2-24**, we import necessary packages for training our classifier:

- matplotlib : For plotting. **Line 3** sets the backend so we can output our training plot to a .png image file.
- keras : For deep learning. Namely, we use the ResNet50 CNN. We will also work with the ImageDataGenerator.
- sklearn : From scikit-learn, we use their implementation of a LabelBinarizer for one-hot encoding our class labels. The train_test_split function segments our dataset into training and testing splits. We also print a classification_report in a traditional format.

- paths : Contains convenience functions for listing all image files in a given path. From there we are able to load our images into memory.
- numpy : Python’s *de facto* numerical processing library.
- argparse : For parsing command line arguments.
- pickle : For serializing our label binarizer to disk.
- cv2 : OpenCV.
- os : The operating system module is used to ensure we grab the correct file/path separator which is OS-dependent.



```
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
    help="path to input dataset")
ap.add_argument("-m", "--model", required=True,
    help="path to output serialized model")
ap.add_argument("-l", "--label-bin", required=True,
    help="path to output label binarizer")
ap.add_argument("-e", "--epochs", type=int, default=25,
    help="# of epochs to train our network for")
ap.add_argument("-p", "--plot", type=str, default="plot.png",
    help="path to output loss/accuracy plot")
args = vars(ap.parse_args())
```

Parsing command line arguments

Our script accepts five command line arguments, the first three of which are required:

- --dataset : The path to the input dataset.
- --model : Our path to our output Keras model file.
- --label-bin : The path to our output label binarizer pickle file.
- --epochs : How many epochs to train our network for — by default, we train for 25 epochs, but 50 epochs can lead to better results.
- --plot : The path to our output plot image file — by default it is named plot.png and placed in the same directory as our training script.

Initializing our labels and loading our data:

```
40 # initialize the set of labels from the spots activity dataset we are
41 # going to train our network on
42 LABELS = set(["weight_lifting", "tennis", "football"])
43
44 # grab the list of images in our dataset directory, then initialize
45 # the list of data (i.e., images) and class images
46 print("[INFO] loading images...")
47 imagePaths = list(paths.list_images(args["dataset"]))
48 data = []
49 labels = []
50
51 # loop over the image paths
52 for imagePath in imagePaths:
53     # extract the class label from the filename
54     label = imagePath.split(os.path.sep)[-2]
55
56     # if the label of the current image is not part of of the labels
57     # are interested in, then ignore the image
58     if label not in LABELS:
59         continue
60
61     # load the image, convert it to RGB channel ordering, and resize
62     # it to be a fixed 224x224 pixels, ignoring aspect ratio
63     image = cv2.imread(imagePath)
64     image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
65     image = cv2.resize(image, (224, 224))
66
67     # update the data and labels lists, respectively
68     data.append(image)
69     labels.append(label)
```

Line 42 contains the set of class LABELS which our dataset will consist of. All labels *not* present in this set will be *excluded* from being part of our dataset. All dataset imagePaths are gathered via **Line 47** and the value contained in args["dataset"] (which comes from our command line arguments).

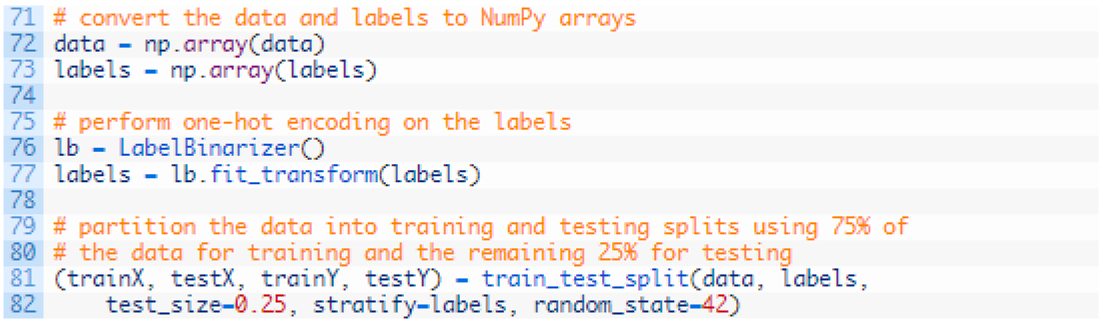
Lines 48 and 49 initialize our data and labels lists.

From there, we begin looping over all imagePaths on **Line 52**.

In the loop, first we extract the class label from the imagePath (**Line 54**). **Lines 58 and 59** then ignore any label not in the LABELS set.

Lines 63-65 load and preprocess an image. Preprocessing includes swapping color channels for OpenCV to Keras compatibility and resizing to *224×224*px. The image and label are then added to the data and labels lists, respectively on **Lines 68 and 69**.

Continuing on, we one-hot encode our labels and partition our data:



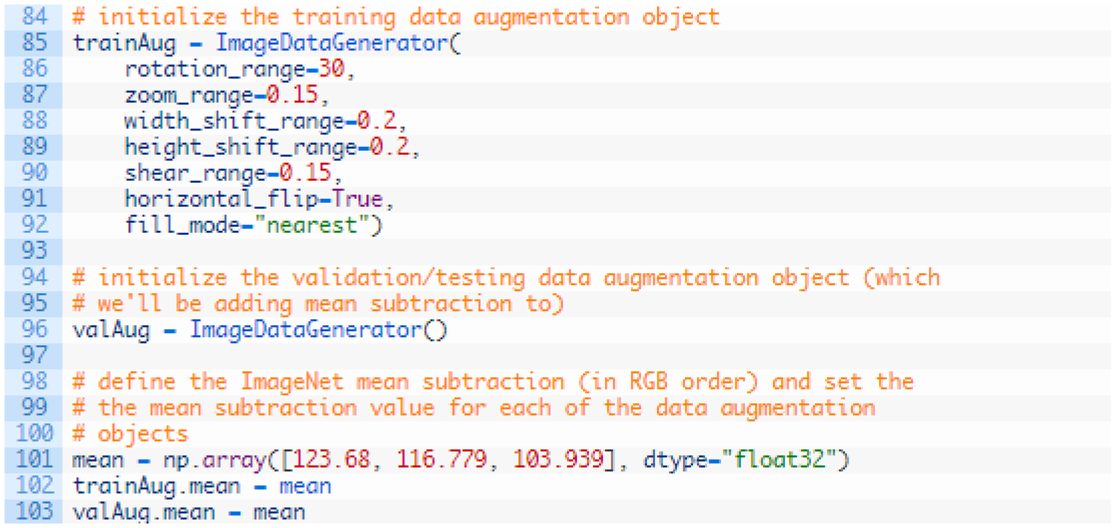
```
71 # convert the data and labels to NumPy arrays
72 data = np.array(data)
73 labels = np.array(labels)
74
75 # perform one-hot encoding on the labels
76 lb = LabelBinarizer()
77 labels = lb.fit_transform(labels)
78
79 # partition the data into training and testing splits using 75% of
80 # the data for training and the remaining 25% for testing
81 (trainX, testX, trainY, testY) = train_test_split(data, labels,
82     test_size=0.25, stratify=labels, random_state=42)
```

Lines 72 and 73 convert our data and labels lists into NumPy arrays.

One-hot encoding of labels takes place on **Lines 76 and 77**. One-hot encoding is a way of marking an active class label via binary array elements. For example “football” may be array ([1, 0, 0]) whereas “weightlifting” may be array ([0, 0, 1]). Only one class is “hot” at any given time.

Lines 81 and 82 then segment our data into training and testing splits using 75% of the data for training and the remaining 25% for testing.

Initializing our data augmentation object:



```
84  # initialize the training data augmentation object
85  trainAug = ImageDataGenerator(
86      rotation_range=30,
87      zoom_range=0.15,
88      width_shift_range=0.2,
89      height_shift_range=0.2,
90      shear_range=0.15,
91      horizontal_flip=True,
92      fill_mode="nearest")
93
94  # initialize the validation/testing data augmentation object (which
95  # we'll be adding mean subtraction to)
96  valAug = ImageDataGenerator()
97
98  # define the ImageNet mean subtraction (in RGB order) and set the
99  # the mean subtraction value for each of the data augmentation
100 # objects
101 mean = np.array([123.68, 116.779, 103.939], dtype="float32")
102 trainAug.mean = mean
103 valAug.mean = mean
```

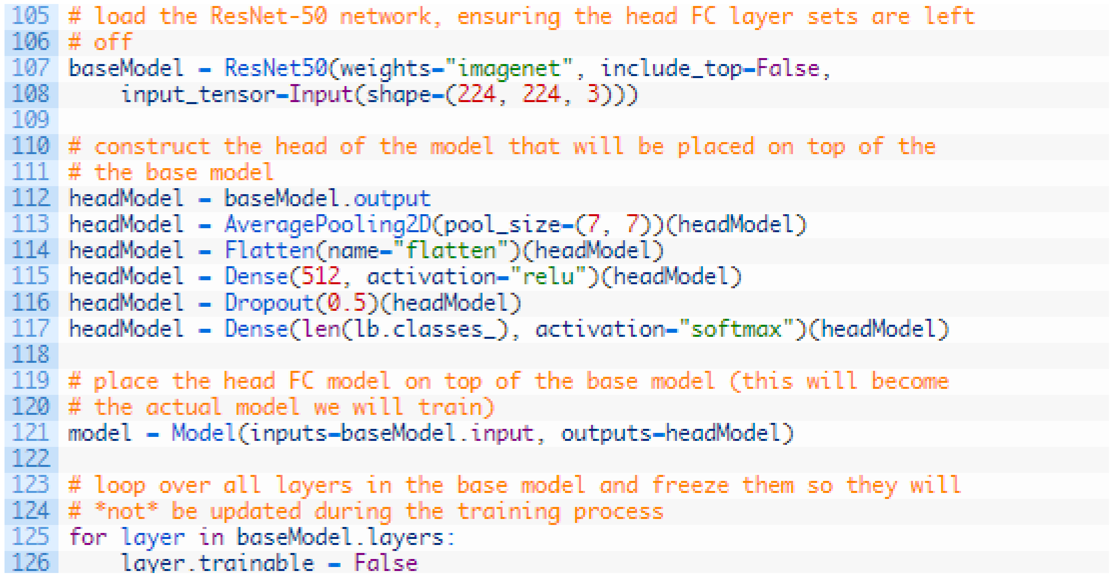
Lines 85-96 initialize two data augmentation objects — one for training and one for validation. Data augmentation is nearly always recommended in deep learning for computer vision to increase model generalization.

The trainAug object performs random rotations, zooms, shifts, shears, and flips on our data. With Keras, images will be generated on-the-fly (it is not an additive operation).

No augmentation is conducted for validation data (valAug), but we perform mean subtraction.

The mean pixel value is set on **Line 101**. From there, **Lines 102 and 103** set the mean attribute for trainAug and valAug so that mean subtraction will be conducted as images are generated during training/evaluation.

Performance of “network surgery” as a part of fine-tuning:



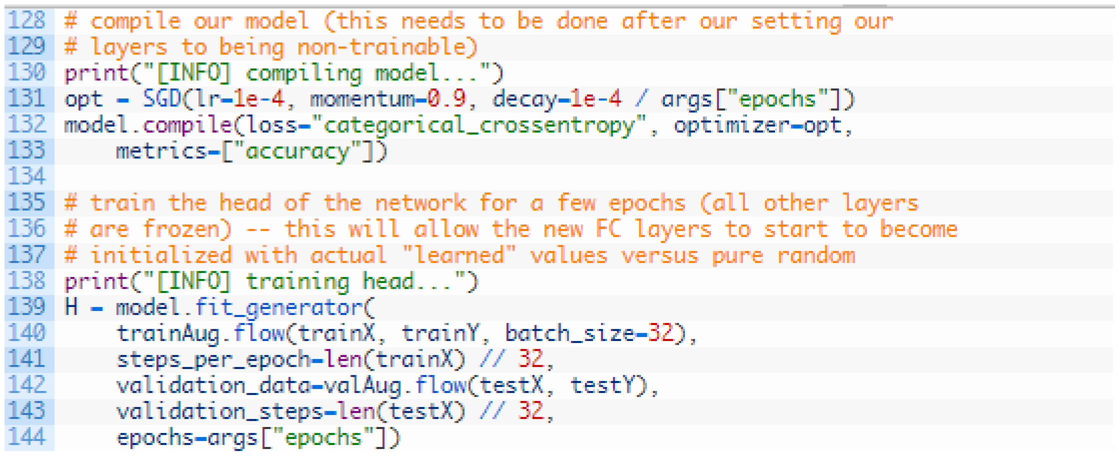
```
105 # load the ResNet-50 network, ensuring the head FC layer sets are left
106 # off
107 baseModel = ResNet50(weights="imagenet", include_top=False,
108     input_tensor=Input(shape=(224, 224, 3)))
109
110 # construct the head of the model that will be placed on top of the
111 # the base model
112 headModel = baseModel.output
113 headModel = AveragePooling2D(pool_size=(7, 7))(headModel)
114 headModel = Flatten(name="flatten")(headModel)
115 headModel = Dense(512, activation="relu")(headModel)
116 headModel = Dropout(0.5)(headModel)
117 headModel = Dense(len(lb.classes_), activation="softmax")(headModel)
118
119 # place the head FC model on top of the base model (this will become
120 # the actual model we will train)
121 model = Model(inputs=baseModel.input, outputs=headModel)
122
123 # loop over all layers in the base model and freeze them so they will
124 # *not* be updated during the training process
125 for layer in baseModel.layers:
126     layer.trainable = False
```

Lines 107 and 108 load ResNet50 pre-trained with ImageNet weights while chopping the head of the network off.

From there, **Lines 112-121** assemble a new headModel and suture it onto the baseModel.

We then freeze the baseModel so that it will ***not*** be trained via backpropagation (**Lines 125 and 126**).

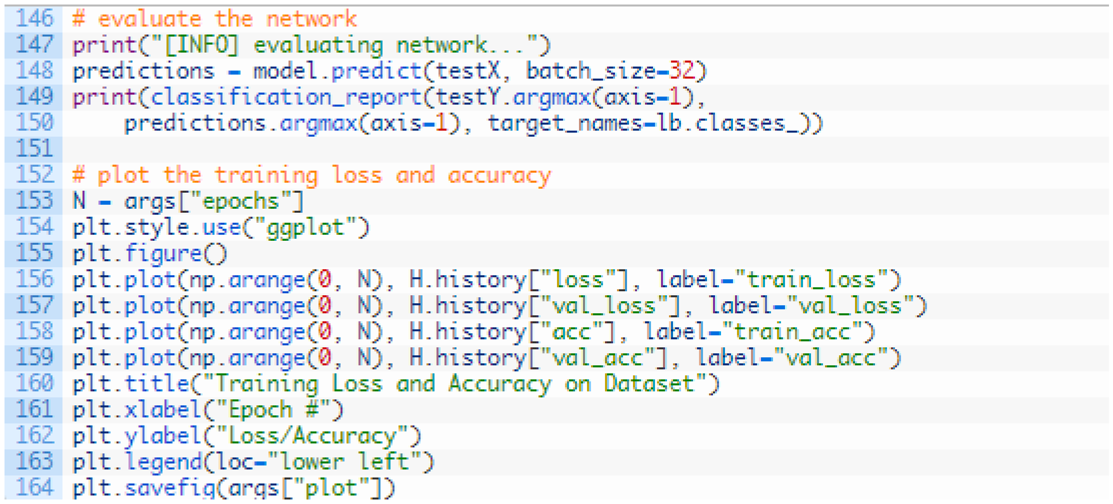
Compiling and training our model:



```
128 # compile our model (this needs to be done after our setting our
129 # layers to being non-trainable)
130 print("[INFO] compiling model...")
131 opt = SGD(lr=1e-4, momentum=0.9, decay=1e-4 / args["epochs"])
132 model.compile(loss="categorical_crossentropy", optimizer=opt,
133     metrics=["accuracy"])
134
135 # train the head of the network for a few epochs (all other layers
136 # are frozen) -- this will allow the new FC layers to start to become
137 # initialized with actual "learned" values versus pure random
138 print("[INFO] training head...")
139 H = model.fit_generator(
140     trainAug.flow(trainX, trainY, batch_size=32),
141     steps_per_epoch=len(trainX) // 32,
142     validation_data=valAug.flow(testX, testY),
143     validation_steps=len(testX) // 32,
144     epochs=args["epochs"])
```

Lines 131-133 compile our model with the Stochastic Gradient Descent (SGD) optimizer with an initial learning rate of 1e-4 and learning rate decay. We use "categorical_crossentropy" loss for training with multiple classes. If we are working with only two classes, be sure to use "binary_crossentropy" loss.
A call to the fit_generator function on our model (**Lines 139-144**) trains our network with data augmentation and mean subtraction.
Since our baseModel is frozen and we’re only training the head. This is known as “fine-tuning”.

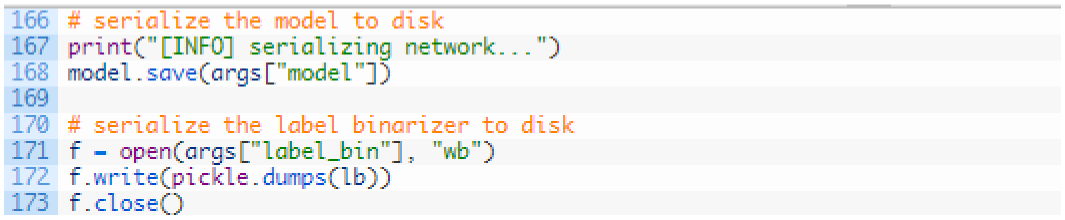
Evaluating our network and plotting the training history:



```
146 # evaluate the network
147 print("[INFO] evaluating network...")
148 predictions = model.predict(testX, batch_size=32)
149 print(classification_report(testY.argmax(axis=1),
150     predictions.argmax(axis=1), target_names=lb.classes_))
151 
152 # plot the training loss and accuracy
153 N = args["epochs"]
154 plt.style.use("ggplot")
155 plt.figure()
156 plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
157 plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
158 plt.plot(np.arange(0, N), H.history["acc"], label="train_acc")
159 plt.plot(np.arange(0, N), H.history["val_acc"], label="val_acc")
160 plt.title("Training Loss and Accuracy on Dataset")
161 plt.xlabel("Epoch #")
162 plt.ylabel("Loss/Accuracy")
163 plt.legend(loc="lower left")
164 plt.savefig(args["plot"])
```

After we evaluate our network on the testing set and print a classification_report (**Lines 148-150**), we go ahead and plot our accuracy/loss curves with Matplotlib (**Lines 153-163**). The plot is saved to disk via **Line 164.**

Finally, we serialize our model and label binarizer (lb) to disk



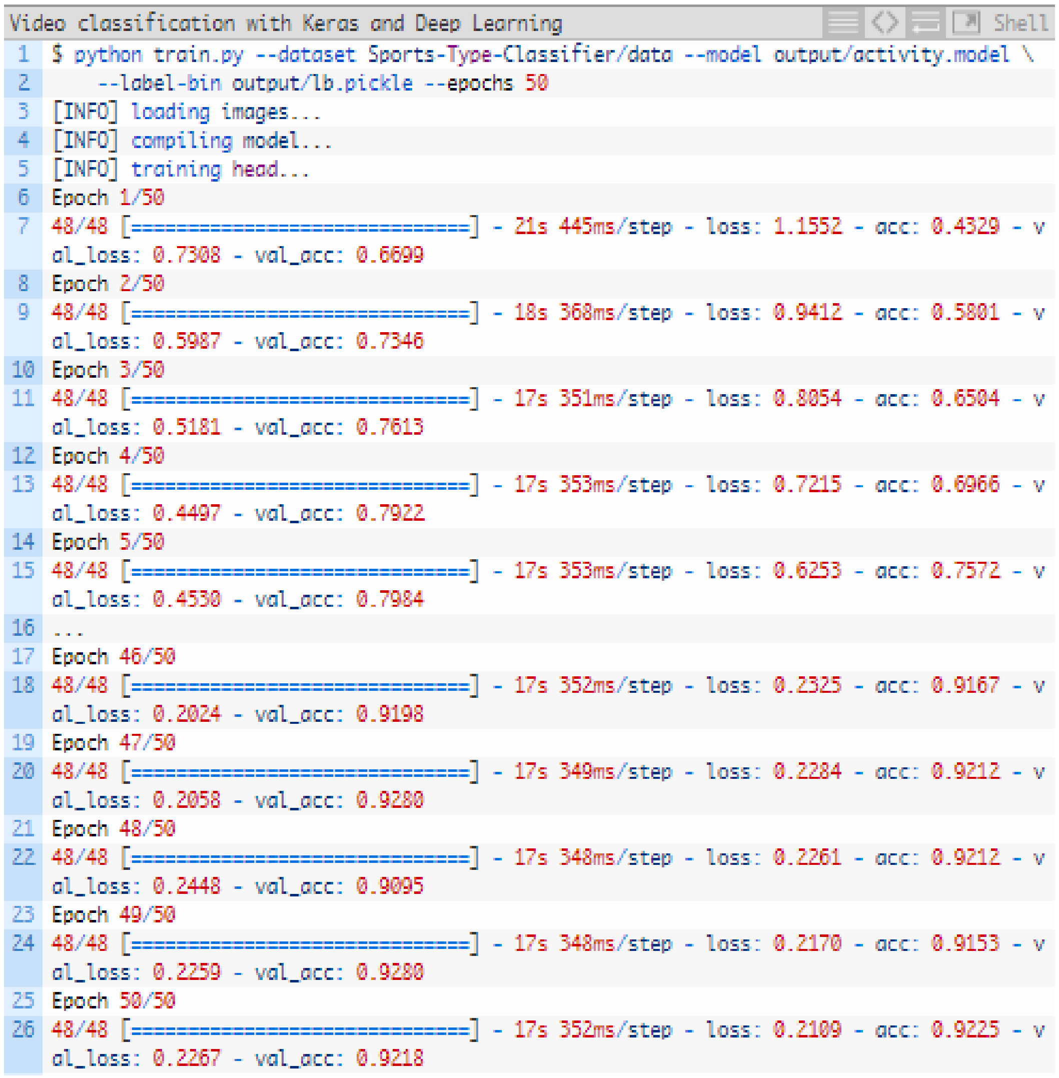
```
166 # serialize the model to disk
167 print("[INFO] serializing network...")
168 model.save(args["model"])
169 
170 # serialize the label binarizer to disk
171 f = open(args["label_bin"], "wb")
172 f.write(pickle.dumps(lb))
173 f.close()
```

Line 168 saves our fine-tuned Keras model.

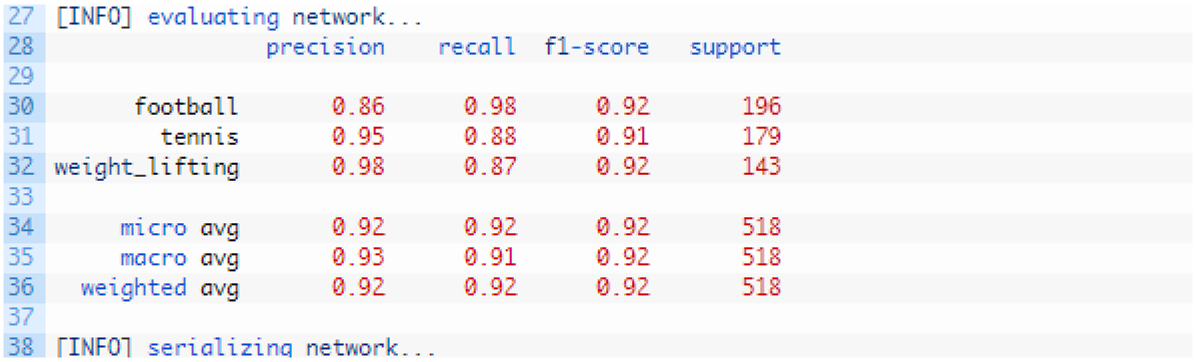
Finally, **Lines 171** serialize and store our label binarizer in Python's pickle format.

10.2 Training Results

Training the model before we (1) classify frames in a video with our CNN and then (2) utilize our CNN for video classification:



```
$ python train.py --dataset Sports-Type-Classifier/data --model output/activity.model \
    --label-bin output/lb.pickle --epochs 50
[INFO] loading images...
[INFO] compiling model...
[INFO] training head...
Epoch 1/50
48/48 [==============================] - 21s 445ms/step - loss: 1.1552 - acc: 0.4329 - v
al_loss: 0.7308 - val_acc: 0.6699
Epoch 2/50
48/48 [==============================] - 18s 368ms/step - loss: 0.9412 - acc: 0.5801 - v
al_loss: 0.5987 - val_acc: 0.7346
Epoch 3/50
48/48 [==============================] - 17s 351ms/step - loss: 0.8054 - acc: 0.6504 - v
al_loss: 0.5181 - val_acc: 0.7613
Epoch 4/50
48/48 [==============================] - 17s 353ms/step - loss: 0.7215 - acc: 0.6966 - v
al_loss: 0.4497 - val_acc: 0.7922
Epoch 5/50
48/48 [==============================] - 17s 353ms/step - loss: 0.6253 - acc: 0.7572 - v
al_loss: 0.4530 - val_acc: 0.7984
...
Epoch 46/50
48/48 [==============================] - 17s 352ms/step - loss: 0.2325 - acc: 0.9167 - v
al_loss: 0.2024 - val_acc: 0.9198
Epoch 47/50
48/48 [==============================] - 17s 349ms/step - loss: 0.2284 - acc: 0.9212 - v
al_loss: 0.2058 - val_acc: 0.9280
Epoch 48/50
48/48 [==============================] - 17s 348ms/step - loss: 0.2261 - acc: 0.9212 - v
al_loss: 0.2448 - val_acc: 0.9095
Epoch 49/50
48/48 [==============================] - 17s 348ms/step - loss: 0.2170 - acc: 0.9153 - v
al_loss: 0.2259 - val_acc: 0.9280
Epoch 50/50
48/48 [==============================] - 17s 352ms/step - loss: 0.2109 - acc: 0.9225 - v
al_loss: 0.2267 - val_acc: 0.9218
```

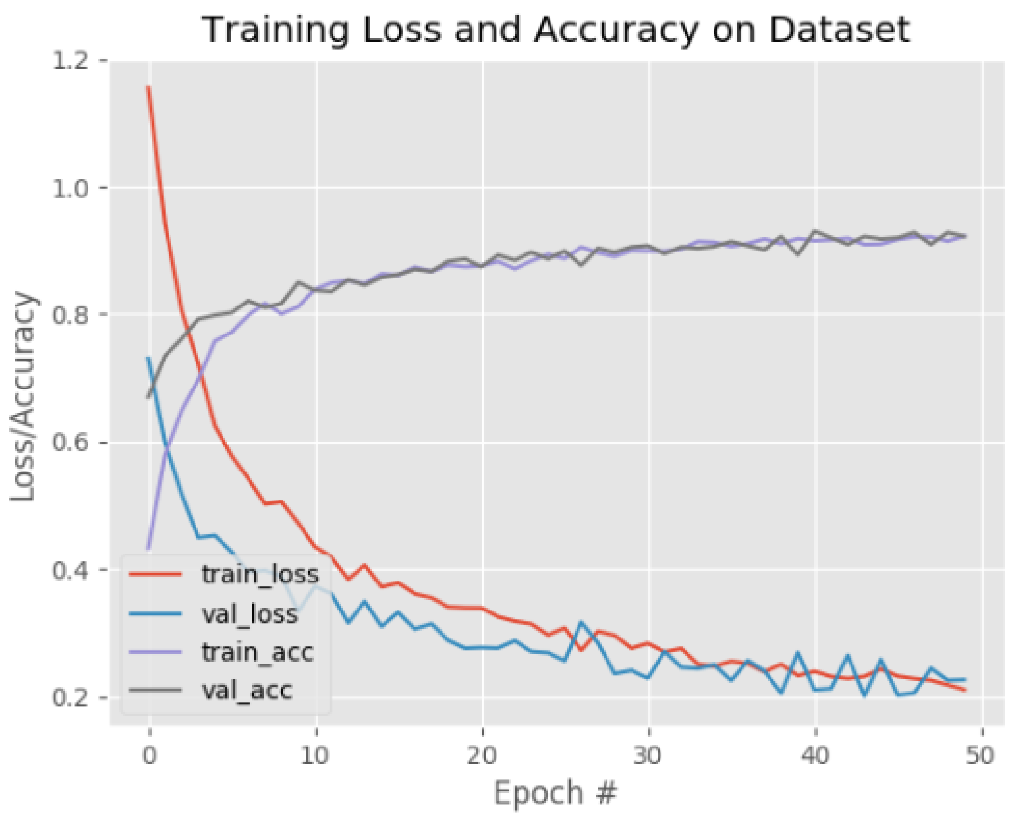


```
[INFO] evaluating network...
                precision    recall  f1-score   support

      football       0.86      0.98      0.92       196
        tennis       0.95      0.88      0.91       179
weight_lifting       0.98      0.87      0.92       143

     micro avg       0.92      0.92      0.92       518
     macro avg       0.93      0.91      0.92       518
  weighted avg       0.92      0.92      0.92       518

[INFO] serializing network...
```



Sports video classification with Keras accuracy/loss training history plot

We are obtaining **~92-93% accuracy** after fine-tuning ResNet50 on the sports dataset.

Checking our model directory we see that the fine-tuned model along with the label binarizer have been serialized to disk:

```
$ ls model/
activity.model  lb.pickle
```

We take these files and use them to implement rolling prediction averaging in the next section.

10.3 Video classification by implementing rolling prediction averaging

To create this script we take advantage of the *temporal nature of videos*, specifically the assumption that *subsequent frames in a video will have similar semantic contents*.
By performing rolling prediction accuracy we are able to "smoothen out" the predictions and avoid "prediction flickering".

```
# import the necessary packages
from keras.models import load_model
from collections import deque
import numpy as np
import argparse
import pickle
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
    help="path to trained serialized model")
ap.add_argument("-l", "--label-bin", required=True,
    help="path to  label binarizer")
ap.add_argument("-i", "--input", required=True,
    help="path to our input video")
ap.add_argument("-o", "--output", required=True,
    help="path to our output video")
ap.add_argument("-s", "--size", type=int, default=128,
    help="size of queue for averaging")
args = vars(ap.parse_args())
```

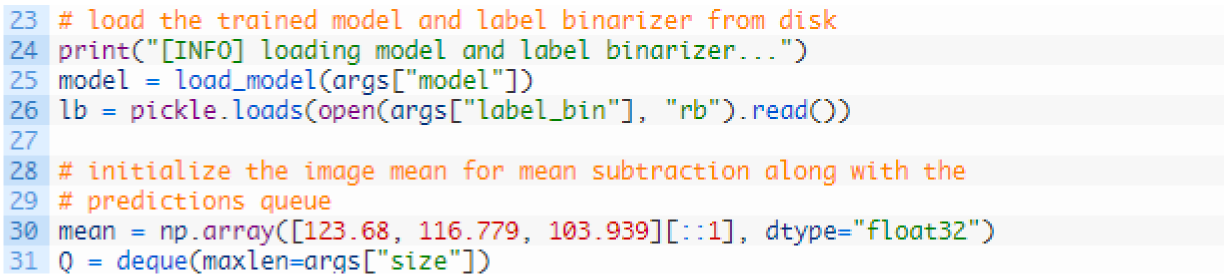
Predict_video.py

Lines 2-7 load necessary packages and modules. In particular, we use deque from Python’s collections module to assist with our rolling average algorithm.

Then, **Lines 10-21** parse five command line arguments, four of which are required:

- --model : The path to the input model generated from our previous training step.
- --label-bin : The path to the serialized pickle-format label binarizer generated by the previous script.
- --input : A path to an input video for video classification.
- --output : The path to our output video which will be saved to disk.
- --size : The max size of the queue for rolling averaging (128 by default). For some of our example results later on, we will set the size to 1 so that no averaging is performed

Initializations:



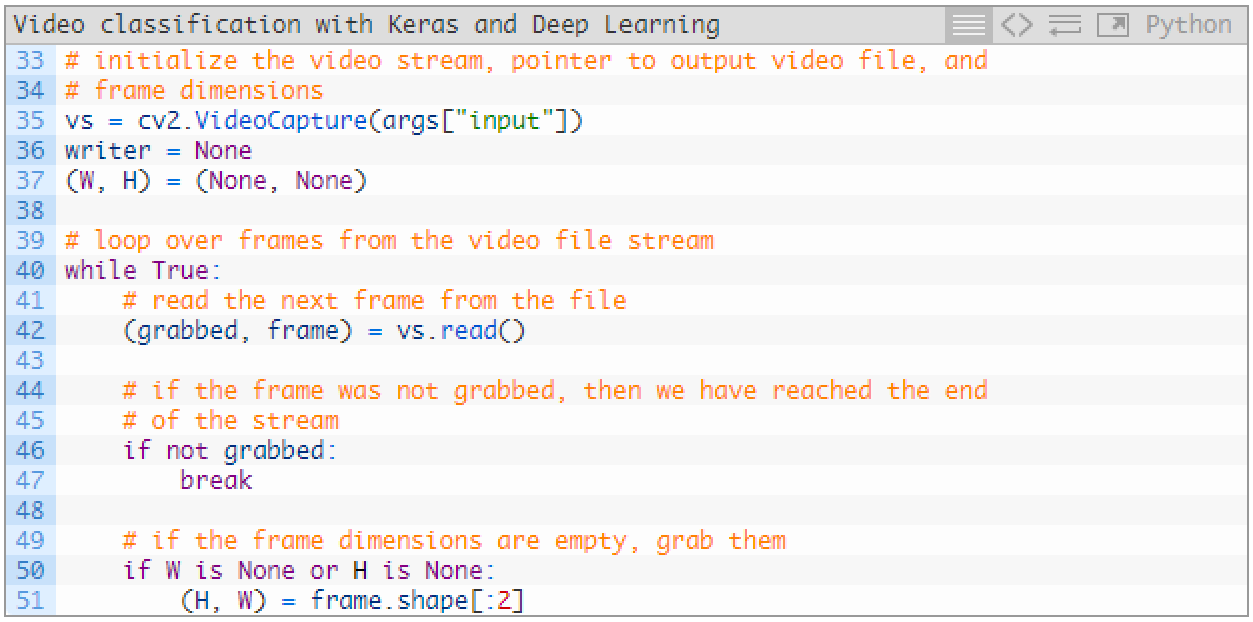
```
23 # load the trained model and label binarizer from disk
24 print("[INFO] loading model and label binarizer...")
25 model = load_model(args["model"])
26 lb = pickle.loads(open(args["label_bin"], "rb").read())
27 
28 # initialize the image mean for mean subtraction along with the
29 # predictions queue
30 mean = np.array([123.68, 116.779, 103.939][::1], dtype="float32")
31 Q = deque(maxlen=args["size"])
```

Lines 25 and 26 load our model and label binarizer.

Line 30 then sets our mean subtraction value.

We use a deque **to implement our rolling prediction averaging.** Our deque, Q , is initialized with a maxlen equal to the args["size"] value (**Line 31**).

Initializing cv2.VideoCapture object and looping over video frames:



```python
33 # initialize the video stream, pointer to output video file, and
34 # frame dimensions
35 vs = cv2.VideoCapture(args["input"])
36 writer = None
37 (W, H) = (None, None)
38 
39 # loop over frames from the video file stream
40 while True:
41     # read the next frame from the file
42     (grabbed, frame) = vs.read()
43 
44     # if the frame was not grabbed, then we have reached the end
45     # of the stream
46     if not grabbed:
47         break
48 
49     # if the frame dimensions are empty, grab them
50     if W is None or H is None:
51         (H, W) = frame.shape[:2]
```

Line 35 grabs a pointer to our input video file stream. We use the VideoCapture class from OpenCV to read frames from our video stream.

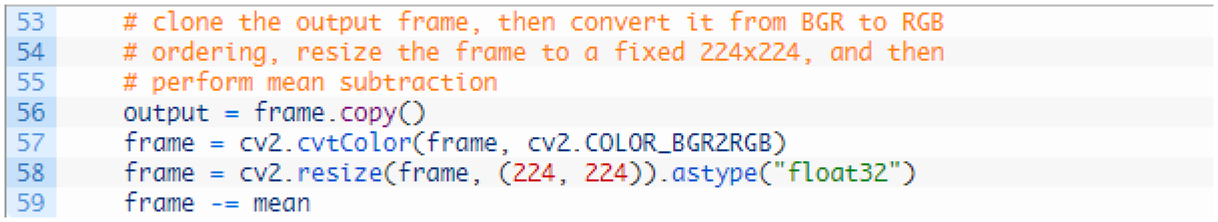
Our video writer and dimensions are then initialized to None via **Lines 36 and 37**.

Line 40 begins our video classification while loop.

First, we grab a frame (**Lines 42-47**). If the frame was not grabbed, then we've reached the end of the video, at which point we break from the loop.

Lines 50-51 then set our frame dimensions if required.

Preprocessing our frame:



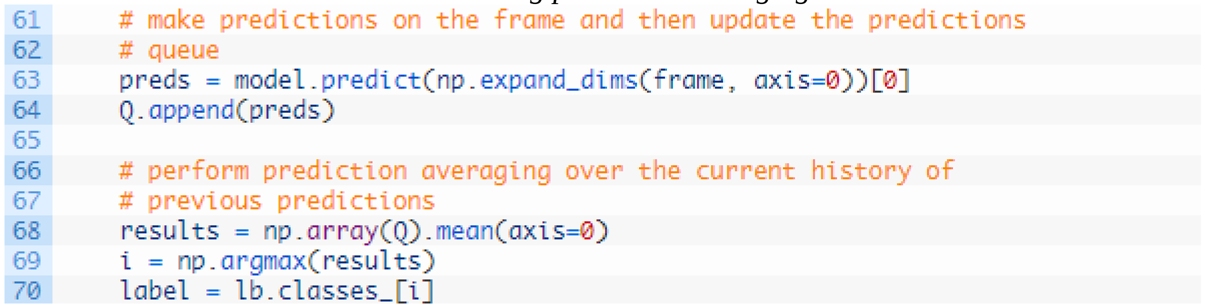
```
53        # clone the output frame, then convert it from BGR to RGB
54        # ordering, resize the frame to a fixed 224x224, and then
55        # perform mean subtraction
56        output = frame.copy()
57        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
58        frame = cv2.resize(frame, (224, 224)).astype("float32")
59        frame -= mean
```

A copy of our frame is made for output purposes (**Line 56**).

We then preprocess the frame using the same steps as our training script, including:

- Swapping color channels (**Line 57**).
- Resizing to *224×224*px (**Line 58**).
- Mean subtraction (**Line 59**).

Frame classification inference and *rolling prediction averaging* **come next:**



```
61        # make predictions on the frame and then update the predictions
62        # queue
63        preds = model.predict(np.expand_dims(frame, axis=0))[0]
64        Q.append(preds)
65
66        # perform prediction averaging over the current history of
67        # previous predictions
68        results = np.array(Q).mean(axis=0)
69        i = np.argmax(results)
70        label = lb.classes_[i]
```

Line 63 makes predictions on the *current* frame. The **prediction results are added to the** Q via **Line 64**.

From there, **Lines 68-70** perform **prediction averaging over the** Q history resulting in a class label for the rolling average. Broken down, these lines find the label with the largest corresponding probability across the average predictions.

Annotating our output frame and writing to disk:

```
72      # draw the activity on the output frame
73      text = "activity: {}".format(label)
74      cv2.putText(output, text, (35, 50), cv2.FONT_HERSHEY_SIMPLEX,
75          1.25, (0, 255, 0), 5)
76
77      # check if the video writer is None
78      if writer is None:
79          # initialize our video writer
80          fourcc = cv2.VideoWriter_fourcc(*"MJPG")
81          writer = cv2.VideoWriter(args["output"], fourcc, 30,
82              (W, H), True)
83
84      # write the output frame to disk
85      writer.write(output)
86
87      # show the output image
88      cv2.imshow("Output", output)
89      key = cv2.waitKey(1) & 0xFF
90
91      # if the `q` key was pressed, break from the loop
92      if key == ord("q"):
93          break
94
95  # release the file pointers
96  print("[INFO] cleaning up...")
97  writer.release()
98  vs.release()
```

Lines 73-75 draw the prediction on the output frame.

Lines 78-82 initialize the video writer if necessary. The output frame is written to the file (**Line 85**). The output is also displayed on the screen until the "q" key is pressed (or until the end of the video file is reached as aforementioned) via **Lines 88-93**.

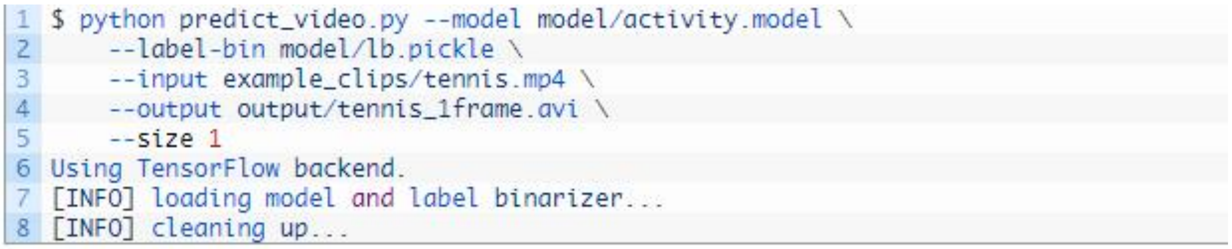
Finally, we perform cleanup (**Lines 97 and 98**).

Json format example of the model after a training epoch:

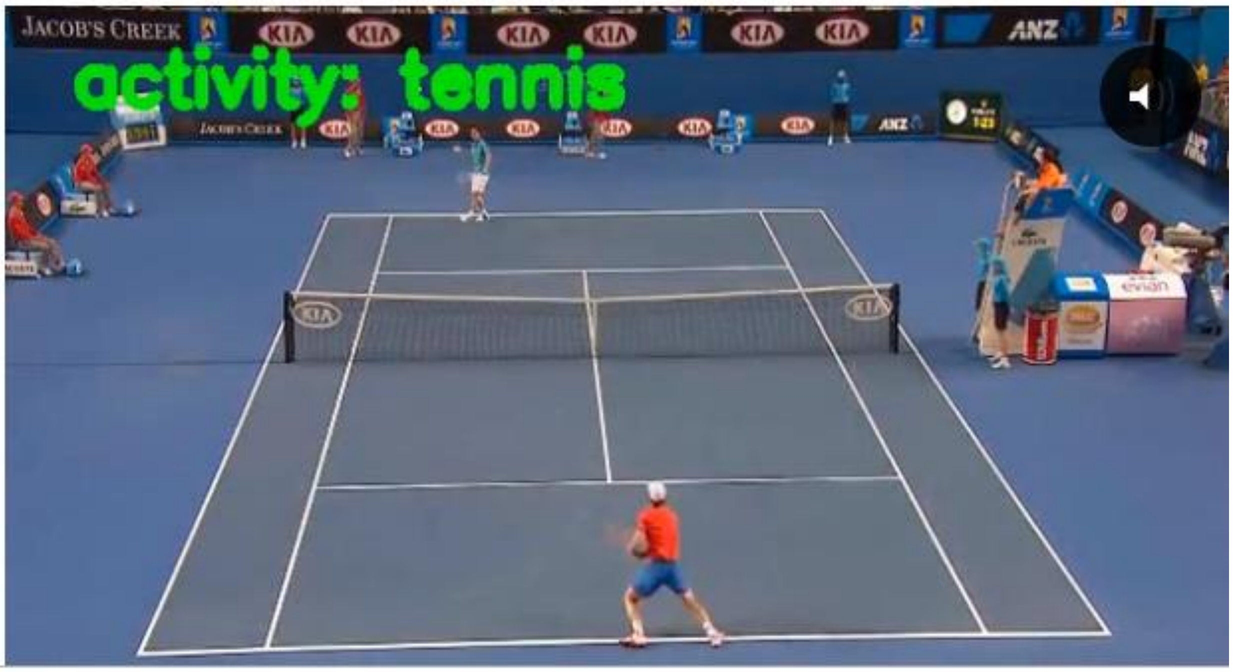
```
{
  "class_name":"Sequential",
  "config":{
    "name":"sequential_1",
    "layers":[
      {
        "class_name":"Dense",
        "config":{
          "name":"dense_1",
          "trainable":true,
          "batch_input_shape":[
            null,
            8
          ],
          "dtype":"float32",
          "units":12,
          "activation":"relu",
          "use_bias":true,
          "kernel_initializer":{
            "class_name":"VarianceScaling",
            "config":{
              "scale":1.0,
              "mode":"fan_avg",
              "distribution":"uniform",
              "seed":null
            }   "keras_version":"2.2.5",
  "backend":"tensorflow"
}
```

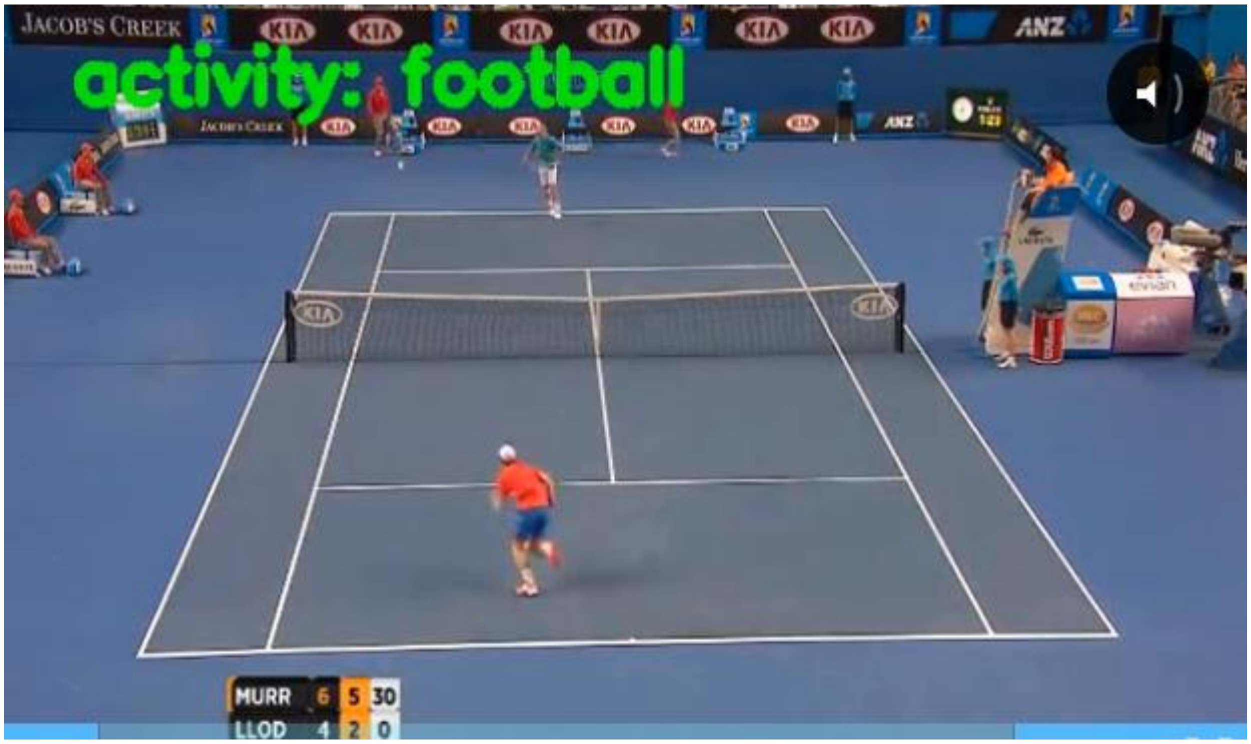
10.4 Video classification results

If we set the --size of the queue to 1, trivially turning video classification into standard image classification:



```
$ python predict_video.py --model model/activity.model \
    --label-bin model/lb.pickle \
    --input example_clips/tennis.mp4 \
    --output output/tennis_1frame.avi \
    --size 1
Using TensorFlow backend.
[INFO] loading model and label binarizer...
[INFO] cleaning up...
```





There is considerable label flickering — our CNN thinks certain frames are “tennis” (correct) while others are “football” (incorrect).

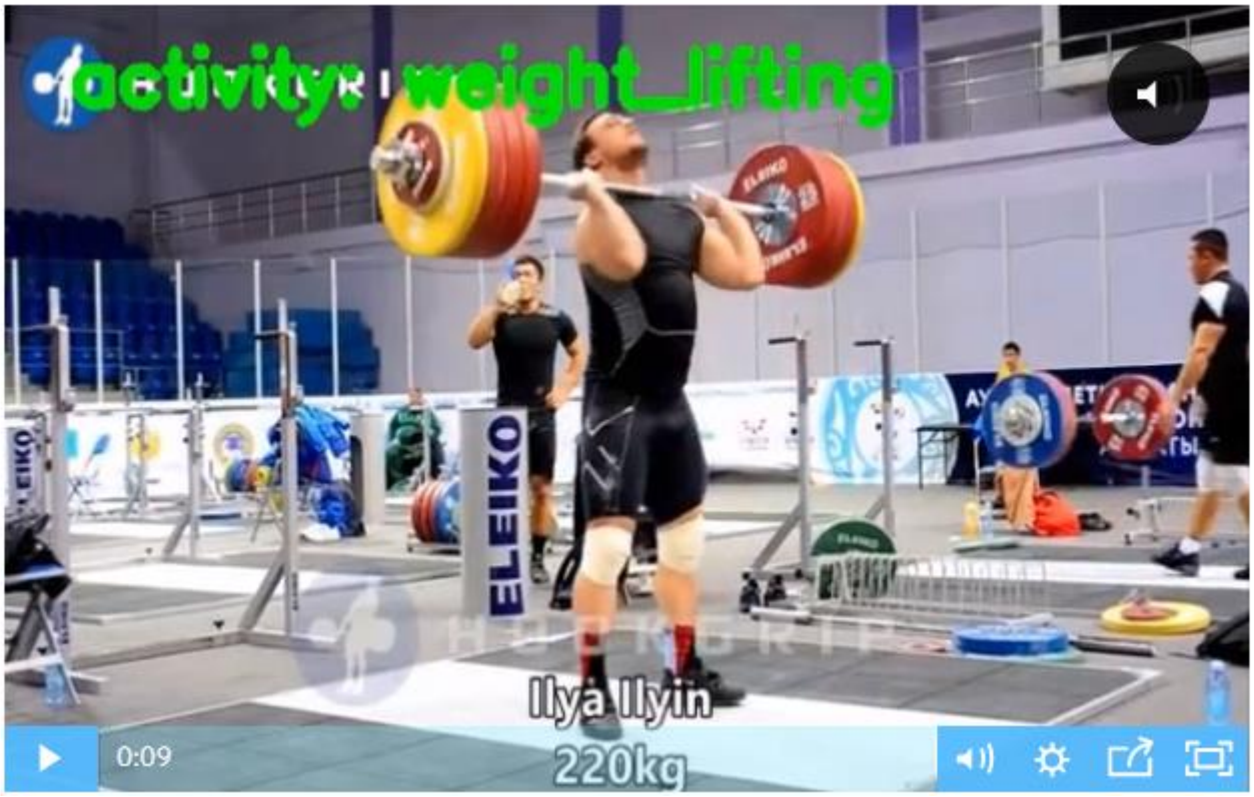
We thus use the default queue --size of 128, thus utilizing our **prediction averaging algorithm** to smoothen the results:

```
$ python predict_video.py --model model/activity.model \
    --label-bin model/lb.pickle \
    --input example_clips/tennis.mp4 \
    --output output/tennis_128frames_smoothened.avi \
    --size 128
Using TensorFlow backend.
[INFO] loading model and label binarizer...
[INFO] cleaning up...
```

Now the CNN has correctly labeled this video as “tennis”, without flickering.

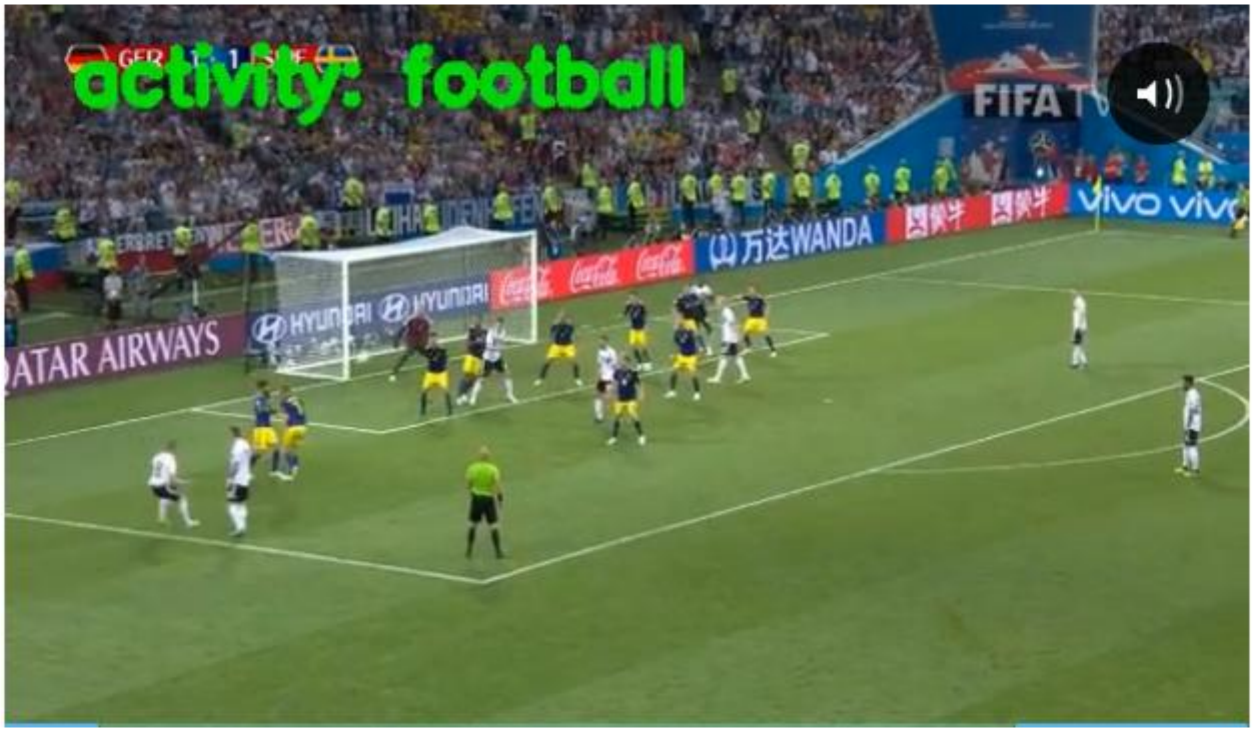
2nd example: Weight-lifting

```
$ python predict_video.py --model model/activity.model \
    --label-bin model/lb.pickle \
    --input example_clips/lifting.mp4 \
    --output output/lifting_128frames_smoothened.avi \
    --size 128
Using TensorFlow backend.
[INFO] loading model and label binarizer...
[INFO] cleaning up...
```



3rd example: Football

```
$ python predict_video.py --model model/activity.model \
    --label-bin model/lb.pickle \
    --input example_clips/soccer.mp4 \
    --output output/soccer_128frames_smoothened.avi \
    --size 128
Using TensorFlow backend.
[INFO] loading model and label binarizer...
[INFO] cleaning up...
```



Hence, we can see that the input video is correctly classified when the queue size is 128,

there is no frame flickering — our rolling prediction averaging smoothens out the predictions.

This algorithm enables us to perform video classification with Keras.

11. Summarization and further work

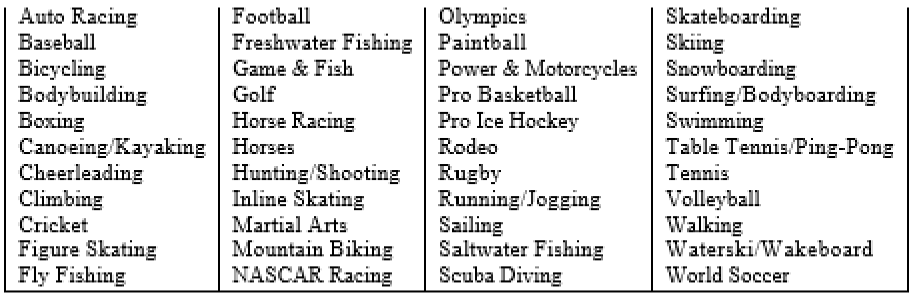
In our project, we learned to perform video classification with Keras and deep learning.

A naïve algorithm to video classification would be to treat each individual frame of a video as independent from the others. This type of implementation will cause “label flickering” where the CNN returns different labels for subsequent frames, *even though the frames should be the same labels.*
More advanced neural networks, including LSTMs and the more general RNNs, can help combat this problem and lead to much higher accuracy. However, LSTMs and RNNs can be dramatic overkill dependent on what task we are performing — **in some situations, simple rolling prediction averaging will give us the desired results.**
Using rolling prediction averaging, we maintain a list of the last *K* predictions from the CNN. We then take these last *K* predictions, average them, select the label with the largest probability, and choose this label to classify the *current* frame. The assumption here is that subsequent frames in a video will have similar semantic contents.
If that assumption holds then we can take advantage of the temporal nature of videos, assuming that the previous frames are similar to the current frame.

The averaging, therefore, enables us to smooth out the predictions and make for a better video classifier.

11.1 Future scope and optimizations

Adapting video classification models for real life advertising tasks requires more work than what was done in this project. If the IAB tech lab content taxonomy is to be used as a base, the network model has to be trained with data from all the categories. Having more categories will decrease the classification performance, as has been shown in the difference between the correct rates of iteration 1 and iteration 2. The taxonomy is currently split into two tiers, with sports being a tier 1 category while football, soccer, ice hockey etc. =- tier 2 categories. In a practical application, one could imagine that both tier categories are put to use. Maybe the network classifies a football clip as soccer but since they are both still under the same tier 1 category the classification is somewhat correct. A person that is interested in one of the tier 2 categories is likely interested in the corresponding tier 1 category as a whole. For example, a person that has shown interest in watching videos belonging to the ”Italian cuisine” tier 2 category is possibly interested in the general field of ”food & drink”, which is the equivalent tier 1 category. In the future the IAB taxonomy is projected to be expanded with further tiers which will increase the chance of misclassifications still being able to apply to the correct audience. One possible example is to have the different types of fishing activities organized under a common category in the sports section. It this case, we could have the tier 3 categories ”fly fishing”, ”saltwater fishing” and ”freshwater fishing” gathered under the tier 2 group ”fishing” in the tier 1 category ”sports”. As seen in the results of the evaluation done in this project, the fishing activities were often confusing for the networks. Adding another tier to the taxonomy could decrease the negative impact of this since the network might be able to classify the video as fishing, even if the category of fishing is wrong. Other possible clustering options for the sport tier are combat sports, team sports, motor sports and extreme sports.



Auto Racing	Football	Olympics	Skateboarding
Baseball	Freshwater Fishing	Paintball	Skiing
Bicycling	Game & Fish	Power & Motorcycles	Snowboarding
Bodybuilding	Golf	Pro Basketball	Surfing/Bodyboarding
Boxing	Horse Racing	Pro Ice Hockey	Swimming
Canoeing/Kayaking	Horses	Rodeo	Table Tennis/Ping-Pong
Cheerleading	Hunting/Shooting	Rugby	Tennis
Climbing	Inline Skating	Running/Jogging	Volleyball
Cricket	Martial Arts	Sailing	Walking
Figure Skating	Mountain Biking	Saltwater Fishing	Waterski/Wakeboard
Fly Fishing	NASCAR Racing	Scuba Diving	World Soccer

The sport categories in the IAB tech lab content taxonomy. IAB is an organization that deals with advertising and marketing and their work includes developing industry standards

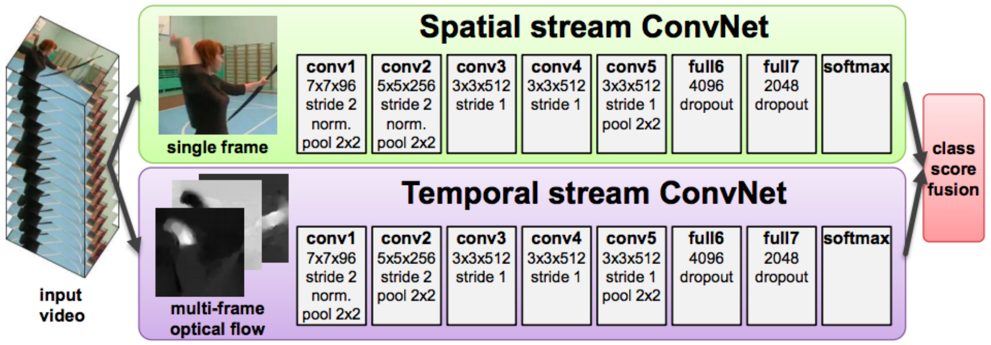
In general, a misclassification of a video in an advertisement streamlining application is not a catastrophe. The possible ”bad” outcome is that the target is presented with products that they do not find interesting. The goal is of course to increase the sales via advertisements as much as possible and this is why a good performance is desired, but in general there is no real harm if a video is misclassified.

To further improve the classification of video clips, other methods such as audio recognition, keyword identification and object detection could also be implemented. By combining these with the general scene classification done in this project, the results could be even better. The video classification itself can also be improved. Besides using a more complex model as base, for example the GoogleNet, there are a couple of things that can be done to augment the performance.

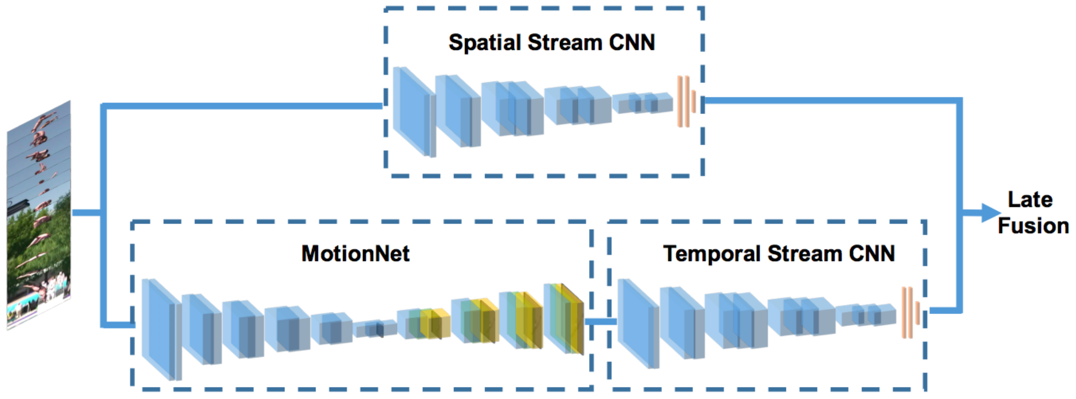
Setting up a workstation with hardware better suited for the task is something that would lower the training and evaluation times drastically. Training on two high end GPUs instead of the CPU I used could do wonders. Combined with a framework that is faster, for example TensorFlow or Torch, the training time could probably be lowered several times.

11.2 Finding conjunctive algorithmic approaches to complement our solution

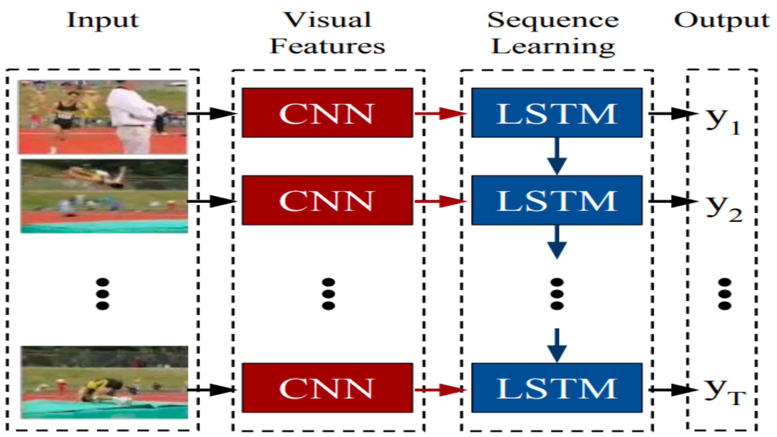
Here we diagrammatically showcase other possible and computationally expensive approaches to augment our project, and describe the best matching algorithm.



1. Two Stream Architecture

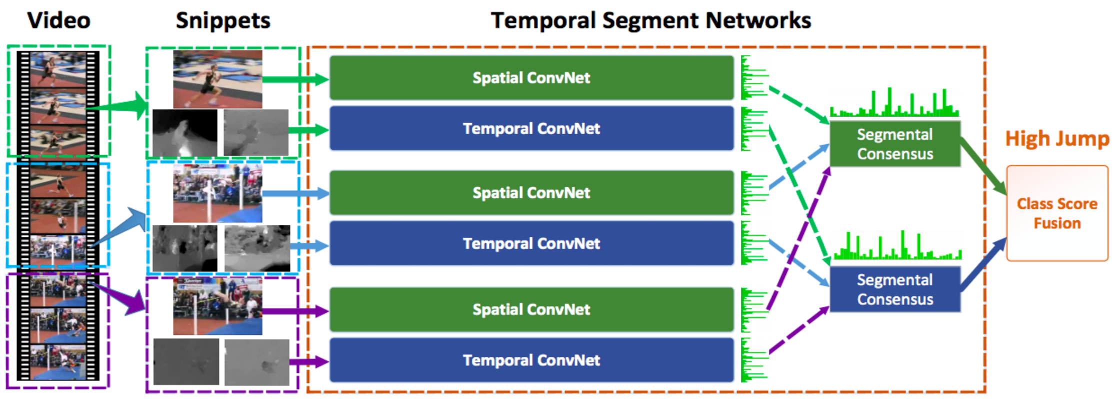


2. Hidden Two Stream-MotionNet which generates optical flow on-the-fly.

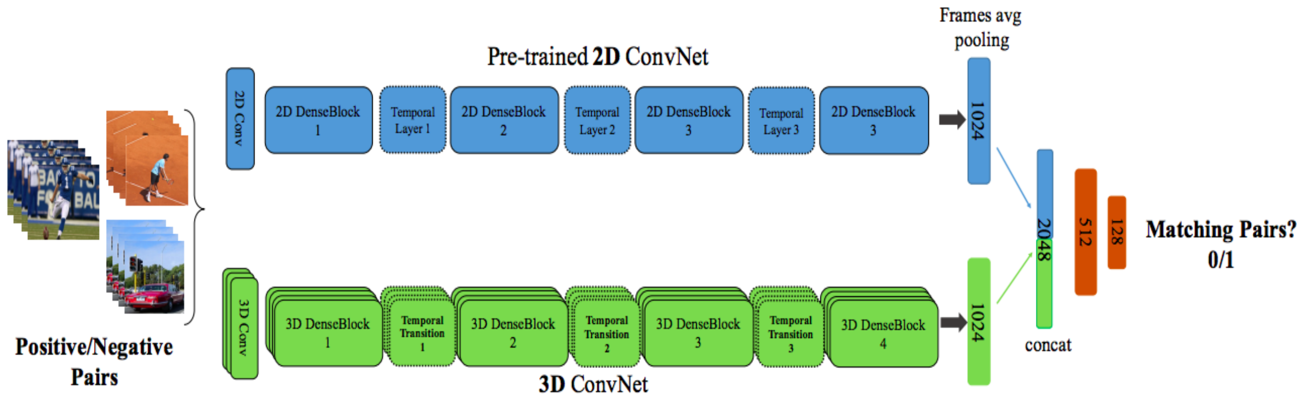


3. Long-term Recurrent Convolutional Networks for Visual Recognition and Description

(Left: LRCN for action recognition. Right: Generic LRCN architecture for all tasks)



4. (TSN)Temporal Segment Network architecture.



5. Transfer learning supervision

6. The best augmentation algorithm for our work:

Conv3D & Attention

Salient features:

• Novel 3D CNN-RNN encoder-decoder architecture which captures local spatiotemporal information

• Use of an attention mechanism within a CNN-RNN encoder-decoder framework to capture global context

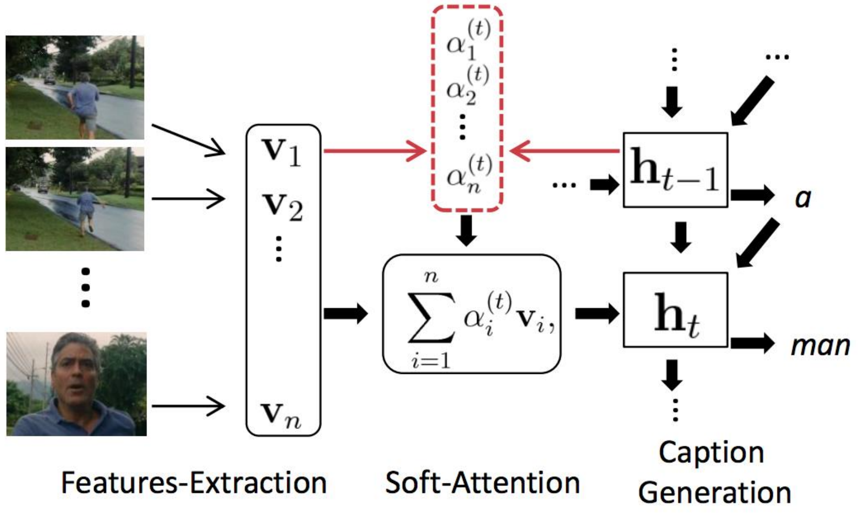
Here we use a 3D CNN + LSTM as base architecture for video description task. On top of the base, we use a pre-trained 3D CNN for improved results.

Algorithm:

The set-up is almost same as encoder-decoder architecture described in LRCN with two differences

1. Instead of passing features from 3D CNN as is to LSTM, 3D CNN feature maps for the clip are concatenated with stacked 2D feature maps for the same set of frames to enrich representation {v1, v2, …, vn} for each frame i. Note: The 2D & 3D CNN used is a pre-trained one and not trained end-to-end like LRCN

2. Instead of averaging temporal vectors across all frames, a weighted average is used to combine the temporal features. The attention weights are decided based on LSTM output at every time step.



Attention mechanism for action recognition

11.3 Key findings

- Based on both theory and the results, it seems like a CNN+LSTM architecture will be well adapted for classifying video clips, if higher accuracy(>95%)is targeted. Else a CNN architecture is fine. A deeper model also provides better performance than shallower ones with the downside that the training and evaluation time increases.
- Using convolutional neural networks is a simple yet efficient way of handling the features of the video clips. It can deal with raw frame data while still having good performance.
- Lowering the size of the input data speeds up the network but lowers the correct rates of the classification. This performance loss can be somewhat compensated by considering more frames per video clip during evaluation.
- Misclassifications of videos in an advertisement applications is not a disaster, but having more tiers in the taxonomy system could possibly alleviate the consequences.
- To further improve the amount of correctly classified videos, other methods such as audio classification, object detection and keyword identification could also be integrated.
- Using a more optimized computer for the networks and using GPUs instead of a CPU is necessary if we want fast and efficient training.

12.1 References

[1] Caffe. http://caffe.berkeleyvision.org/. Accessed: 2019-09-26.

[2] Codemill. https://codemill.se/. Accessed: 2019-09-23.

[3] Deeplearning4j. http://deeplearning4j.org/. Accessed: 2019-09-26.

[4] Github Repository. https://github.com/bdtkarlsson/VideoClassificationThesisProject.

[5] IAB (interactive advertising bureau). https://www.iab.com/. Accessed: 2016-09-23.

[6] IAB tech lab content taxonomy. https://www.iab.com/guidelines/iab-tech-lab-content-taxonomy/. Accessed: 2016-09-23.

[7] Smart Video. http://smartvideo.io/. Accessed: 2016-12-06.

[8] Sports-1M-Dataset. https://github.com/gtoderici/sports-1m-dataset/blob/wiki/ProjectHome.md. Accessed: 2016-09-26.

[9] TensorFlow. https://www.tensorflow.org/. Accessed: 2016-09-26.

[10] Deeplearning4j. Convolutional networks. http://deeplearning4j.org/convolutionalnets.html. Accessed: 2016-09- 23.

[11] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *CVPR*, 2015.

[12] Douglas M. Hawkins. The problem of overfitting. *J. Chem. Inf. Comput. Sci*, 44:1–12, 2004.

[13] Kaiming He. Deep Residual Learning. http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf. Accessed: 2016-11-07.

[14] Lee Jacobson. Introduction to artificial neural networks - part 1. The Project spot: http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7, 2013-12-05. Accessed: 2016-09-23.

[15] WildML. Understanding convolutional neural networks for nlp. http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/, 2015-11-07. Accessed: 2016-09-23.

[16] Joe Yue-hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. Technical report, University of Maryland, College Park and University of Texas at Austin and Google, Inc, 2015

12.2 Appendix

Source code link:

https://drive.google.com/drive/folders/16aKtmNayohGyBKF4S2b72zICXhYwtMPx?usp=sharing

Github repository for main files:

https://github.com/rishavgiri6/FinalYearProject