UNIVERSITY OF TECHNOLOGY OF BELFORT-MONTBÉLIARD

Deep Learning for egocentric vision

ST50 thesis - P2020

GUETARNI Bilel

Computer Science department
Image, Interaction et Réalité Virtuelle

Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud
Avenue des Sports 29
1400 Yverdon-les-Bains
heig-vd.ch

Company tutor
PEREZ-URIBE Andres

UTBM tutor
GABER Jaafar

Contents

Glossary

egocentric view First person view. 6, 8

jointly trained Training several deep learning systems as one unique system. 9

Natural Language Processing Automatic process of language by computers. 7, 9, 16

synthetic-to-real domain gap The unreal property of synthetic data that impacts models accuracy when trained on. 7

vanishing gradient Trend of the gradient to vanish during backpropagation in deep architectures. 16

Acknowledgements

I would like to express my deep gratitude to the Haute École d'Ingénierie et de Gestion du Canton de Vaud (HEIG-VD) for receiving me during this internship and particulary to the International Office for providing me support, dedication and availability in these particular times. I would also like to thank the Communauté du Savoir that makes these extraterritorial cooperation possible that allow students to benefit from other universities education. Finally, I am particularly grateful to Mr. Andres PEREZ-URIBE and all his team, for all the advices, support and teaching they gave me. I adress them my very great appreciation for the assistance and the kindness they showed to me and the time they consecrated for my project.

1. Introduction

1.1 Institut des Technologies de l'Information et de la Communication

The *Institut des Technologies de l'Information et de la Communication* (IICT) is the biggest research institute of the HEIG-VD university with more than 50 researchers and engineers and near 20 teachers. Its research is divided into software engineering, telecommunications, artifical intelligence, digital security and communication systems. They realize each year more than 60 projects, mostly with industrials.

In this institut work the professor Andres PEREZ-URIBE and his team, the Intelligent Data Analysis group, on big data and intelligent data analysis. This team "works on making sense of data by developing and using data-driven models using Machine Learning approaches (including artificial intelligence, artificial life and bio-inspired approaches) as an alternative to analytical models for classification, prediction, explanation and visualization". They "aim at providing solutions to deal with the current scenario of "data deluge" (Big Data) and to provide innovative solutions for the new ubiquitous computing world (Internet of Things and Quantified Self)". They developed several applications based on intelligent systems as:

- Agrovision: an aerial image based tool for agriculture monitoring.
- ClusterSITG: an automatic tool for metadata organization.
- CrowdStreams: an application for crowd analyze and monitoring.

1.2 Internship context

Since the availability of huge databases, modern deep learning has been applied on several tasks with amazing performance. One of such tasks is action recognition. There have been considerable research on this topic using deep learning, but the lack of relatively large scaled datasets have limited the possibilities to develop supervised learning solutions. Recently, there have been considerable works to create such datasets, launching the exploration

of deep learning architectures for action recognition. There exist a dataset, centred on actions performed in kitchen environments where the objective is to recognize a performed action (in a video) in a first person view, also called egocentric view. The purpose of this project is to create a deep learning system that is able to recognize the actions present in the dataset with the highest accuracy possible.

We will first describe the relationship between deep learning and computer vision to understand which important part the first has became to the latter. Then, we'll review the egocentric action recognition state-of-the-art in the literature to describe afterwards which methods were studied. In a critical process, we then critic the obtained results along with an interpretation. Finally, we'll conclude with some observations and difficulties encountered and open some roads for future investigation.

The code is available at: `https://github.com/bguetarni/Egocentric-activity-recognition`.

2. Action recognition in egocentric vision

2.1 Deep Learning and Computer Vision

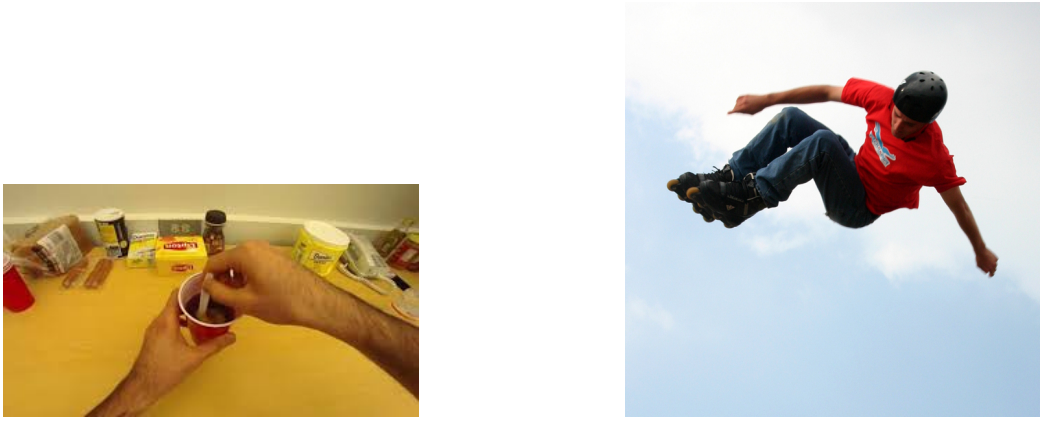
Deep Learning is a really successful field that is currently under innumerable investigations. Despite the well-known difficulties associated, numerous people jump into it every day. It has been successfully applied to several domains like Computer Vision, Natural Language Processing (NLP), board and video games and a raft of others. The first one is currently the most explored with a lot of academic research associated with huge industrial applications. Deep learning has revolutionized the field of computer vision by its accuracy never reached by classical algorithms. From the beginning, computer vision played an important role in the development of deep learning [LeCun et al., 1989, LeCun et al., 1998, Fukushima, 1981]. However, it suffers from a major drawback, that is its need of enormous datasets. As the tasks tackled become more and more complex, the systems need more and more data to reach an acceptable accuracy. Also, most systems for computer vision are trained in a supervised learning fashion which requires data to be labeled, and for some tasks it is really hard to find such ones (even if synthetic data can be used, this still raise the problem of synthetic-to-real domain gap). Moreover, the more data we have, the more computations are then needed, and we know that recent deep learning systems take a long time to train.

Despite those drawbacks, and because several workarounds have been developed, deep learning is still seen as a powerful tool to enhance our systems.

Action recognition

Content recognition is at the heart of the deep learning literature since the beginning. It began with numbers [Fukushima, 1981, LeCun et al., 1998], animals (e.g. cats and dogs) and more recently complex concepts such as actions. Human action recognition requires developing systems that can capture motion over time (sequence prediction), or systems that can recognize an action through a single frame. The first case has seen the emergence of new ar-

chitectures as 3D convolutional networks [Ji et al., 2012] and two-stream architectures [Sudhakaran et al., 2019, Wang et al., 2016, Ye et al., 2019] (see 2.2.1). It is worth to notice that action recognition is itself split in two categories: first person view, also known as egocentric view (Figure 1a), and third person view (Figure 1b); in this work, we focus on egocentric view action recognition. If we look in the literature, we'll see that egocentric vision has been neglected over third person view for a long time. Nonetheless, large scaled recently published datasets have exposed the egocentric view problematic and encouraged research for. A similar task that is increasingly



(a) First person.

(b) Third person.

Figure 1: Action views.

covered is image captioning [Xu et al., 2015], where a system try to add a caption to an image; which is related to scene understanding. It is easy to see that they are sensibly similar and actually, many advances that came out for image captioning are now used for action recognition (attention mechanism, recurrent neural networks).

Some applications

Among the different applications of action recognition we find patient follow-up. Medical staff could use automatic egocentric action recognition to find which actions in the daily-life of a person with physical abilities disorders are the most difficult for them[1]. And then use these analyses to conduct a targeted rehabilitation process.

[1]Of course recognize the action is not sufficient, one should detect the disorders beforehand.

Another interesting potential of action recognition systems is video audio description, this could make all TV programs or videos accessible in audio description for blind or visually impaired people.

2.2 State-of-the-art

In this chapter we introduce the egocentric action recognition state-of-the-art. We'll describe a particular architecture that a majority of papers have focused on; for a specific reason. Furthermore, we'll see how different approaches, currently under investigation, taken from Natural Language Processing have achieved considerable performance.

2.2.1 Two-stream architectures

Currently, the most used architecture for action recognition is the so-called two-stream architecture. It consists of two subsystems, each one processing different image modalities.

Optical flows

As said in the introduction of this chapter, to recognize an action it is almost necessarily to capture the motion. This problematic of describing motion has been extremely present in the computer vision literature and it gave rise to optical flows, sometimes also called optical flow fields. An optical flow is, most of the time, computed as a shift between two gray images. Two types of optical flows exist: *sparse* and *dense*. Sparse optical flows track distinct pixels among the images using feature selectors (e.g. Harris corner detector) and computes their displacement. On the other hand, dense optical flows are a per-pixel-motion estimate method. Here, we'll consider the latter one; see Appendix C for some examples. There are several methods to compute dense optical flows, but the most known (and used) are the TV-L1 [Pérez et al., 2013] and the Gunner Farnebäck [Farnebäck, 2003] algorithms.

Nowadays, even deep learning is considered for modeling optical flows [Hur and Roth, 2020].

Literature

In [Wang et al., 2016], two Convolutional Neural Networks (CNN or ConvNet, they'll be used interchangeably but describe the same thing) are jointly trained; rigorously speaking, one stream is trained before and used to initialize the other one before being jointly trained. The two-stream are identical

except for the number of features expected for the input. The first one expect RGB images, which makes 3 features, while the second one expect stacked optical flows (2 or more features). Random RGB images and optical flows are drawn from the action video and fed to their corresponding CNN. At then end, a consensus function combines all the outputs to infer the final prediction. Notice that as the RGB images are processed individually, the motion can hardly be encoded by the RGB stream, in opposite with the optical flow stream. They also show that using RGB differences rather than stacked RGB images improves the performance.

Yet, processing the images separately prevent the model to use the temporal correlation of the images. Even if we use optical flows, they describe an infinitesimal motion compared to the whole action motion. In [Ye et al., 2019], this idea of using the temporal correlation of the images makes use of Recurrent Neural Networks (RNN) through a ConvLSTM [Shi et al., 2015]. As originally developed, the Long Short-Term Memory (LSTM), that is discussed in 2.3.3, takes as input a sequence of vector and output a single vector. However, as we consider images it would be a waste to shrink the images into vectors for feeding them to a LSTM; because it would result in a loss of information. To overcome this lost of information when using LSTM, [Shi et al., 2015] developed the ConvLSTM cell, that allows to process images inside a LSTM. For this end, they replace the matrix product by a convolution. This simple yet effective trick create a sort of Convolutional Recurrent Neural Network.

Later on, [Ye et al., 2019] used a ResNet-101 pre-trained on ImageNet as feature extractor. As the others, this ConvNet is duplicated to use RGB frames and optical flows. The resulting feature maps of the two modes are then fused using different strategies. A ConvLSTM processes the sequence of fused feature maps before a final fully-connected layer.

Last but not least, Sudhakaran et al. [Sudhakaran et al., 2019] pointed a major difficulty to action recognition in egocentric vision that is the "huge variations present in the data caused by the highly articulated nature of the human body". Along with this difficulty, they make the following assumption (that surely makes sense): "the discriminative information is often confined locally in the video frame [...] thus, not all convolutional features are equally important for recognition". From these, they derive an architecture that uses a widely studied mechanism nowadays, attention. Simply put, they add a learnable mapping to weight the input feature maps before feeding them to the cell. Additionally, the output gate uses a weighted version of the cell state rather than the sequence element. They incorporate this mechanism with a ConvLSTM and achieve the best results to this date.

2.2.2 EPIC-KITCHENS

What would be deep learning without data ?

From the very beginning, people have been struggling to push deep architectures into configurations with acceptable accuracy. Researchers claim that the more data is gathered for training, the likeliest the model will converge. However, certain problems require labels that are either complex or long to get. For example, image segmentation requires that every pixels of an object is labeled within its category, and this, for all objects in every images. On the other hand, action recognition requires to associate an action for every frame of video; we can still get around this by associating labels to a timelapse which contains an action. This reflects the difficulty to produce accurate and large scaled datasets for supervised learning tasks.

For egocentric action recognition, a team of researchers have published a large scaled dataset of kitchen actions in egocentric vision: EPIC-KITCHENS. This dataset has been first published in 2018, [Damen et al., 2018], with approximately 40K actions using more than 55 hours of recording (EPIC-KITCHENS-55). Recently, a second version came out, [Damen et al., 2020], with more than 90K action segments which have been extracted from 100 hours of recording (EPIC-KITCHENS-100) splitted into train (75%), validation (10%) and test (15%) sets. Not only it contains action recognition labels, but also action detection, action anticipation and several others challenges. To this date, it's the biggest open egocentric action recognition dataset available.

Each action is delimited by its frames and timesteps in the video, and is labeled with a *verb* and a *noun* (see Figure 2a). The biggest advantage of

Figure 2: Close (verb) the refrigerator (noun).

using this dataset is its natural looking data. The videos were recording by researchers, engineers, students and other people, in their kitchens while they were doing daily activities like: cooking, cleaning, etc. The resulting data

have a very natural aspect which makes it suitable to train models with real world environments knowledge. However using a big dataset implies a finest management of the memory (see 2.6.1). A major drawback that natural data contained is the biased proportion between labels. For any activity, all actions that we do are not uniformly distributed in time. This is illustrated by the proportion of samples of each verb class in EPIC-KITCHENS-55 in Figure 3; the same problem occur for the noun classes. This class imbalance produces

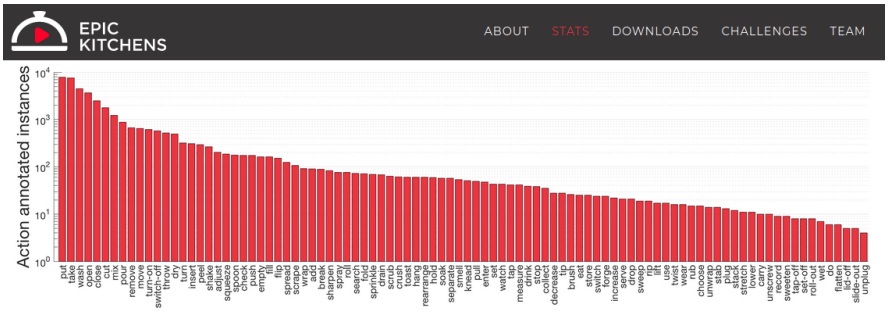


Figure 3: EPIC-KITCHENS 55 verbs instances.

highly biased models toward the most represented classes in the data. and is still present in the new version. We looked for using *class weights*, that associate weights to each sample according to its class, but this didn't help but it resulted in a lot of instability for the models. In our experiments we first used EPIC-KITCHENS-55 to reduce the time for hyperparameters search and one final model were trained on EPIC-KITCHENS-100 (2.4).

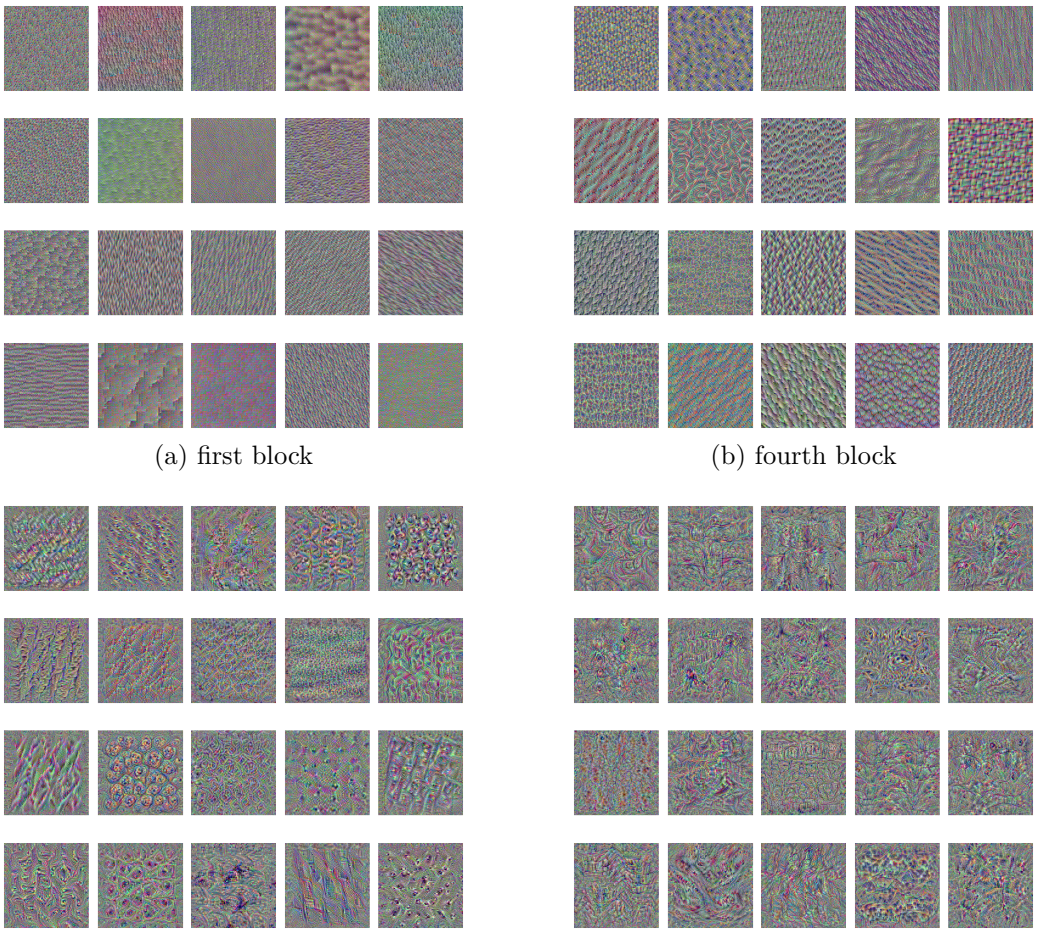
2.3 Studied methods

2.3.1 Transfer Learning

As ConvNet architectures have become bigger and bigger over the past 20 years, the time and amount of computations to train these networks grew exponentially. Therefore, people searched for a way to reduce this training time and eventually recycle the learned knowledge, by transferring them from a domain to another. It's called Transfer Learning. The main argument of this practice is that big ConvNets layers, when trained on big dataset, will learn different features depending on their position in the architecture. Early layers tend to learn basic features such as: edge detectors, Gabor filters, etc. And the deeper we go in the architecture, the more complex these features become. To proof this, we provide what is called a *feature visualization* of

different kernels of MobileNetV2 in Figure 4; the reason why MobileNetV2 is given later.

Feature visualization



(a) first block (b) fourth block

(c) eighth block (d) twelfth block

Figure 4: MobileNetV2 kernels visualization.

In order to provide insight of what a deep ConvNet has learned, [Olah et al., 2017] developed the feature visualization tool. Based on gradient ascent, this technique produces an image that will maximize the activation of a specific neuron (i.e. output of a single kernel convolution). We can do this because: "neural networks are, generally speaking, differentiable with respect to their inputs. If we want to find out what kind of input would cause a certain behavior — whether that's an internal neuron firing or the final output behavior — we

can use derivatives to iteratively tweak the input towards that goal." It is useful to visualize which features a kernel, and more generally a layer, is looking for, because it provides information of the role of this layer in the architecture.

Already before, [Erhan et al., 2009] applied this principle of activation maximization on different architectures trained on MNIST. They found that some of the resulting images "look like pseudo-digits". When trained on natural images, the first layer filters were similar to "Gabor-like features". Combined with other results, they suggest that "higher level units did indeed learn meaningful combinations of lower level features". Knowing this, we can assume that the early features of a ConvNet are common features to every sort of task. Then we can reuse these trained models knowledge for other tasks; this is what transfer learning is all about, transfering knowledge from a domain to another.

We provide such feature visualization for the first, fourth, eihgth and twelfth blocks, respectively, of MobileNetV2 in Figure 4; a block is a serie of convolutions, batch normalization and activation function. We clearly see that in block 1 (4a), the features are edge detectors associated with high frequency patterns. Deeper, in block 4 (4b), we see more complex features, but still, they seem to be a basic combination of the previous basic features. And if we go even deeper, in blocks 8 and 12 (4c and 4d respectively), the observed features are far more complex and are probably revealing what type of features one can encounter in ImageNet images.

Based on these visuals and the previous assumptions, we first attempted to use the 9 first blocks of MobileNetV2, pre-trained on ImageNet, as feature extractor for our architectures. We decided to do so because the last blocks of such deep networks are too much specialized on the features present in ImageNet, that are different from those in EPIC-KITCHENS. We provide in appendix A, examples of both datasets to see how different they are. Even if basic features are shared (edges, corners...), we thought that the high-level features to extract are not transferable from ImageNet to EPIC-KITCHENS.

MobileNetV2 [Sandler et al., 2019] is a ConvNet developed with the objective to optimize the computations for edge device utilization. Modern deep networks have millions of parameters, and they are trained with modern technologies like GPU and TPU. However, embedding such networks in devices with limited computation power implies to minimize the computations, especially for real-time solutions. MobileNetV2 is a deep ConvNet made of several residual blocks, similarly to ResNet [He et al., 2015], which has been shown to help with vanishing and exploding gradient. It has the advantage of decomposing convolutions in two stages, first a depthwise convolution which processes one convolution per channel, and then a pointwise

convolution that combines the channels. This reduces the computations by almost a factor of k^2, where k is the kernel size. It contains a total of 16 blocks with an incremental number of features (24, 32, 64, ..., 320). The final convolution, after the last block, outputs 1280 features. From the beginning of the project, we intended to, if a model was successful enough, use it in an edge device with real-time predictions. Thus, we designed our architectures using MobileNetV2 pre-trained on ImageNet as a feature extractor.

2.3.2 Architectures base

All the different architectures that we tried share a common base (drawn in Figure 5a):

1. We draw 20 uniformly spaced RGB images ($N = 20$) from the action video.
2. Each image is fed to the pre-trained MobileNet V2.
3. The resulting feature maps are then fed sent to a temporal processing subsystem; a spatial pooling must be applied beforehand in some cases.
4. Finally, we use two fully-connected layers to output the *verb* and *noun* probability distributions.
5. The error backpropagation is applied to the fully-connected layers and the temporal processing only (red box in Figure 5a).

We use the cross-entropy error between predicted and ground-truth labels as error function.

$$L_{verb} = -\sum_i y_i^{verb} log(\hat{y}_i^{verb})$$

$$L_{noun} = -\sum_i y_i^{noun} log(\hat{y}_i^{noun})$$

$$Loss = L_{verb} + L_{noun}$$

where y and $\hat{y}$ are the ground-truth and model prediction respectively.

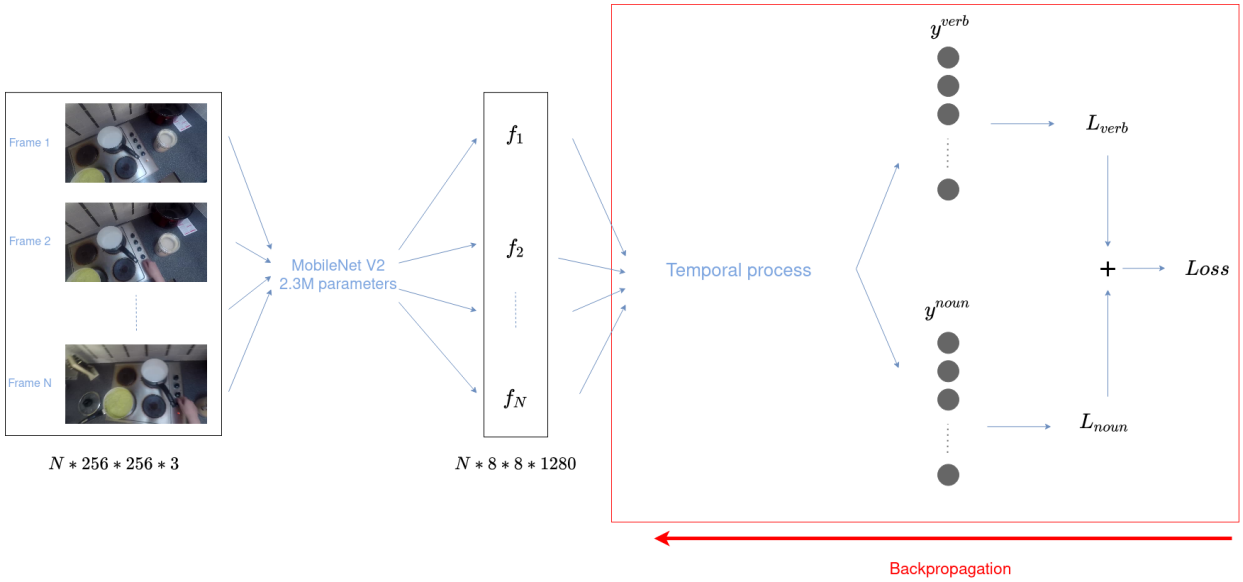


Figure 5: Common base.

2.3.3 Temporal process subsystem

LSTM

Recurrent neural networks have been largely successfully applied to sequence-to-sequence prediction problems, especially in Natural Language Processing [Wu et al., 2016, Shen et al., 2018, Miao et al., 2020]. Their ability to propagate information during the inference is particularly appreciated as well as their independence of the sequence length, indeed the weights are shared across the sequence elements (which required the introduction of the *backpropagation through time* algorithm). They also attracted a lot of attention due to their ability to counter vanishing gradient, especially LSTM [Hochreiter and Schmidhuber, 1997]; a brief recall of how RNN and LSTM work is described in appendix B. Despite their success, LSTM are also known to encounter chaotic behaviour sometimes [Bertschinger and Natschläger, 2004, Laurent and von Brecht, 2016], which make them difficult to train with little data. Albeit these difficulties, we still tried to use LSTM because of its popularity. Nonetheless, we must ensure that what is fed to the LSTM is a sequence of vectors and not images. We thus used *global average pooling* on the output of MobileNetV2 to shrink them to vectors. Global average pooling is illustrated in Figure 6a, where k is the mean filter with the same size as A.

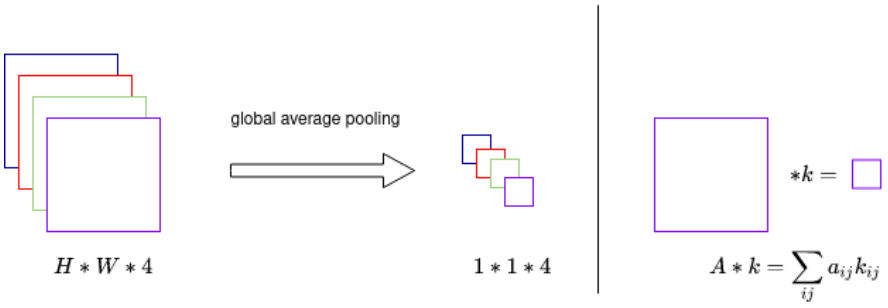


Figure 6: Global average pooling.

ConvLSTM

As mentioned in 2.2.1, ConvLSTM were designed to counter the inability of LSTM to work with images. Indeed, LSTM works with vectors only, which constrain us to shrink the images (or feature maps) into vectors, wether by flattening or applying spatial pooling or feature extraction. However, these operations result in a loss of information, sometimes to the detriment of useful information. ConvLSTM is a good alternative that comes with all the potential of a CNN fused with the recurrence ability of LSTM. However, one must consider that by using images, we drastically augment the number of parameters needed in the ConvLSTM. As an example if we consider a sequence of $N*256*256$ RGB images going through MobileNetV2, we recover a sequence of $8 * 8$ feature maps with 1280 channels each one. To send this through a LSTM, one must beforehand apply a spatial pooling to shrink those feature maps into $1*1$. This sequence of vectors ($N*1*1*1280 \Leftrightarrow N*1280$) is then fed to a LSTM which output a feature vector with, let's say, 50 features. This LSTM will then have 266K parameters. If we do the same with a ConvLSTM, meaning we don't have to apply spatial average beforehand, that takes the $N*8*8*1280$ feature maps and use 50 kernels of size $4*4$ with a stride of 4 (as an example), it will result in an output of size $2*2*50$, thus requiring to apply spatial pooling before any fully connected layer; however applying spatial pooling on a $2*2$ result in less information loss than on $8*8$. This ConvLSTM will unfortunately have 4.2M parameters, which is more than 15 times bigger than the LSTM, while both result in a feature vector of dimension 50.

To compare them, we provide a graphical illustration, Figure 7, of different values such as:

- *time*: the time required to infer on one sample (milliseconds)
- *sequence length*

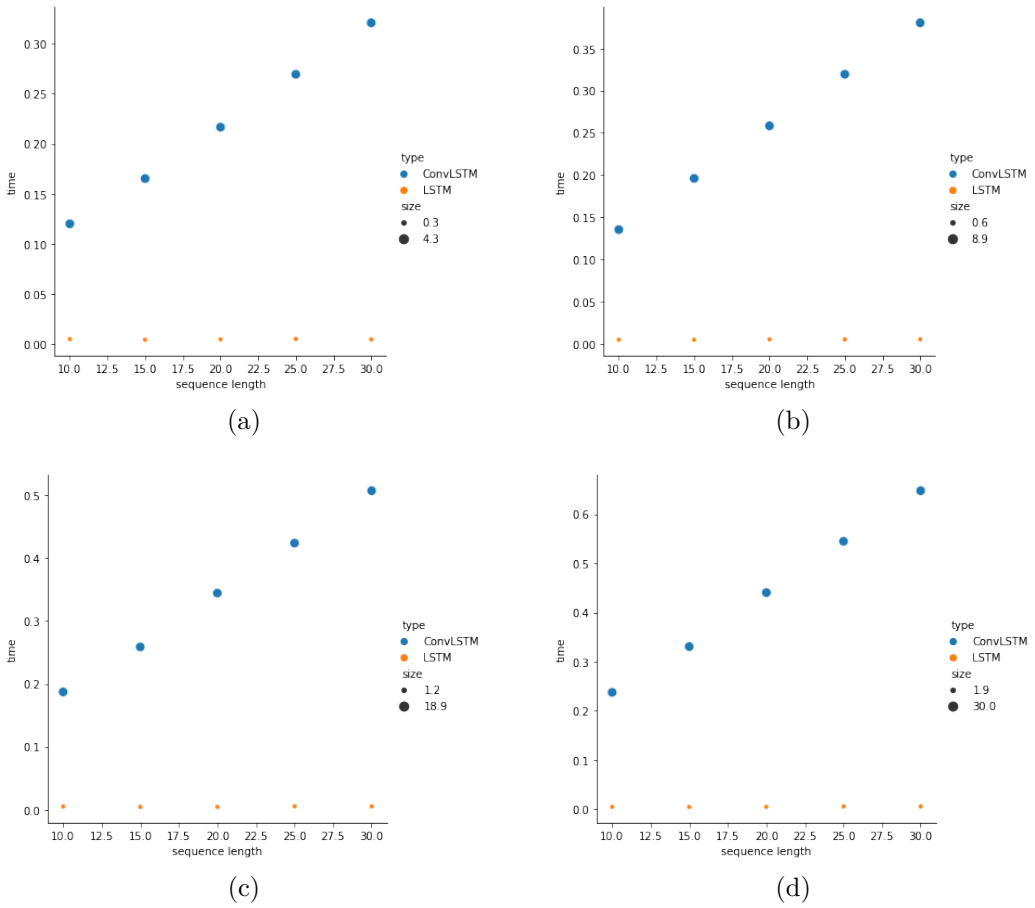


Figure 7: Comparison of differents aspects between LSTM and ConvLSTM.

- *size*: number of parameters (rounded to the million)

The computations were done on Google Colab using an NVIDIA Tesla K80. The Figure 7a represent the computation time as a function of the sequence length, sequences of $8 * 8 * 1280$ images (or feature maps), where the output dimension is 50. Figures 7b, 7c and 7d are similar but with an output dimension of 100, 200, and 300 respectively. We can see that the advantage of the ConvLSTM over LSTM is rapidly taken over by the computation time and number of parameters to train. In our studies, we used a ConvLSTM layer with 8 kernels of size $4 * 4$. We also chosen to use a stride of 4, to avoid losing too much during the spatial pooling afterwards.

Except for the convolutions, the internal structure of ConvLSTM is the same as LSTM (see appendix B).

ConvNet1D

Before using these RNN architectures to manipulate time-series data, a different approach was to use 1D convolutions, also called *temporal convolutions*. It consists of taking a sequence of feature vectors and applying 1D convolutions. The kernel size of the convolution define the time window perceived by the convolution, i.e. how many seconds (or milliseconds) of the action the convolution is looking at. The advantage of temporal convolutions is the possibility to decompose the temporal processing in many depth levels. Rather than taking the kernel size being equal to the sequence length, which enforce it to extract the temporal correlations in one step, we can decompose the extraction in many depth levels as illustrated in Figure 8. On the left we have a 1 level deep temporal convolution which have to extract the temporal correlations in one convolution, whereas on the right we increase the depthness to construct a hierarchical extraction of the temporal correlations; notice that the circles are actually vectors here.

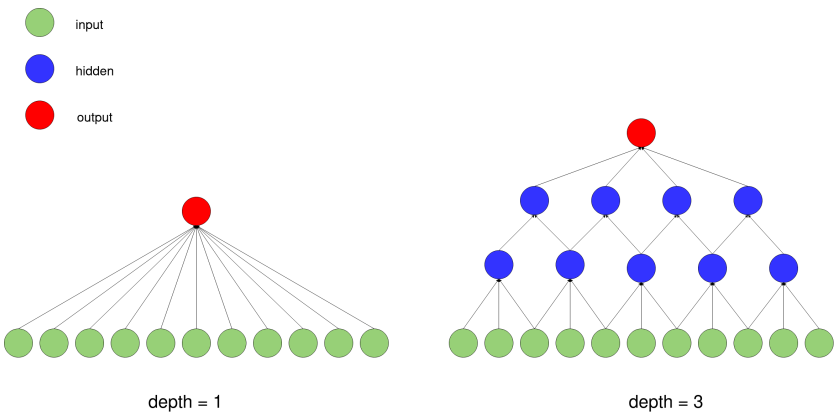


Figure 8: Temporal convolutions of different depth.

We use these temporal convolutions the same way we use a LSTM, i.e. the images go through MobileNetV2, then we retrieve the resulting feature maps, we apply some spatial pooling to shrink them to vectors, and feed the resulting feature vectors to a CNN of 1D convolutions. In our studies, this *ConvNet1D* consists of stacked 1D convolutions (using kernels of size 3) with batch normalization and LeakyReLU activation.

2.4 Results and interpretation

The models were trained during 100 epochs, using the Adam optimizer with an initial learning rate of 0.001 and learning rate decay with a factor of 0.2

starting from epochs 35 and every 20 epochs. In most cases the training was stopped before the end because of the performance not improving or overfitting (early stopping). The batch size was fixed to 32 when using two GPUs and to 16 when using only one. We also added a dropout of 0.5 before the two fully-connected layers to reduce overfitting. These hyperparameters were fixed because they provided the best results after several experiments, we will not report the intermediate results here because they were really poor and not extremely different.

In a first attempt, we simplified the problem in order to explore the different possibilities in terms of architecture and hyperparameters. For this, we used a third of the data of EPIC-KITCHENS-100 (that is more or less similar to EPIC-KITCHENS-55 in term of size) to reduce the training time; we still split it in training and validation sets. Furthermore, rather than training the models with all the classes (97 verbs and 300 nouns) we decided to take profit of the categorization of these classes illustrated in Figure 9; notice that the data imbalance problem is still present even if the classes are encapsulated in categories. As the results were extremely poor, we went

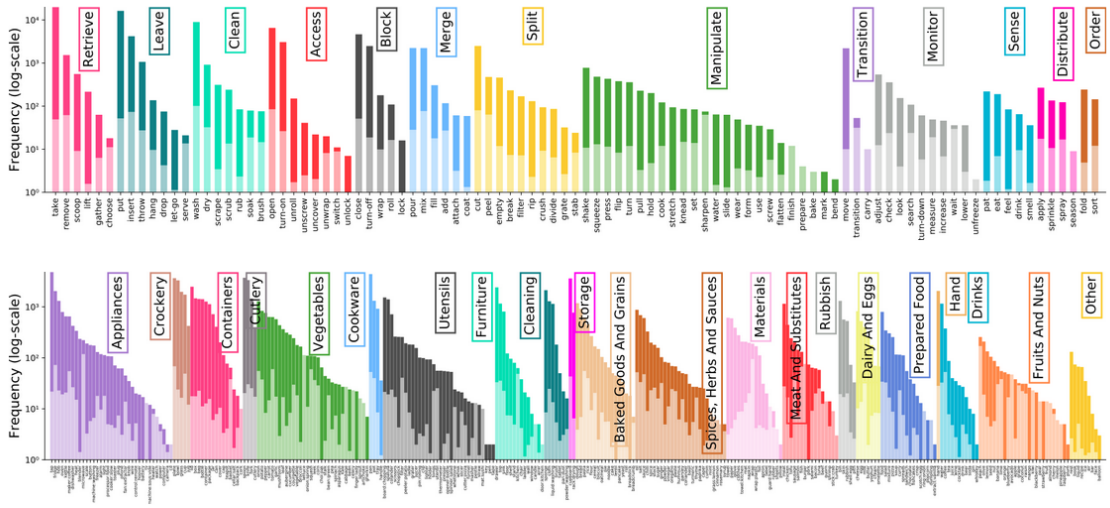


Figure 9: EPIC-KITCHENS 100 categories (top verbs / bottom nouns).

further in the simplification by considering only the verbs. From there, we noticed a real improvement, so we report the accuracy of the three subsystems on the validation set in Table 1; we couldn't use the test set because the labels are not available at this date. We see that the top-1 accuracy is actually quite poor for all of the temporal subsystems but with a noticeable difference between ConvNet1D and LSTM, the first is more than two times better than the latter. When looking at the training curve of the ConvLSTM

subsystem	top-1(%)	top-3(%)	top-5(%)
LSTM	7.11	21.26	31.58
ConvLSTM	12.00	35.95	58.3
ConvNet1D	**18.17**	**50.56**	**73.69**

Table 1: Acurracy of the temporal subsystems on verb categories.

subsystem, Figure 10, we could think that it generally predicts well on the training data. But, looking at its confusion matrix on the validation set,

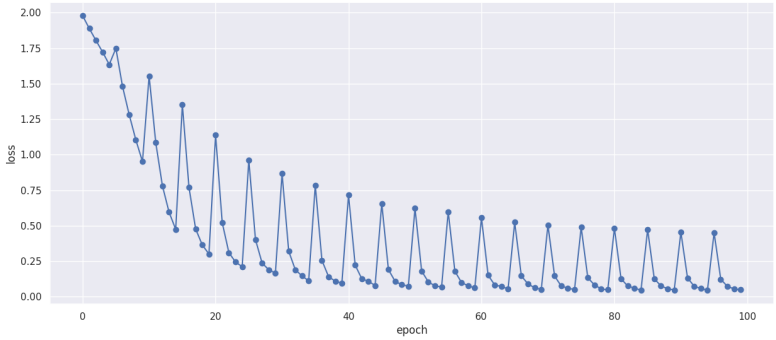


Figure 10: ConvLSTM training loss curve.[2]

Appendix E-17, we realize that it predicts more or less one class (category). We observe the same phenomena for the LSTM, Figure 11 and Appendix E-18. We suspect that the models learned to predict only a restricted number of classes that are ubiquitous in the dataset compared to the other ones, which can mean that they were stucked at a local minima of the loss function. This is a difficult situation to dealt with and an extensive research of the hyperparameters could lead to better results. From our perspective, we think that the poorness of the results illustrates the difficulty of the task and the difficulty to learn on natural data that present high statistic variability.

In a final attempt, we trained a model using ConvNet1D, we would do the other ones if we had enough time, on EPIC-KITCHENS-100 using verbs and nouns without categorizing. We report the results on the validation set in Table 2. The results are lower than before because of the bigger number of classes and the fact that we also predict the nouns. In the third column we see the accuracy of the action, where we consider the actions as all the possible pair (verb,noun), which produces $97 * 300 = 29100$ classes. However, we are not surprised by the results because if the model can poorly perform

[2]The weird peaks in the curve are due to the training process described in 2.6.1.

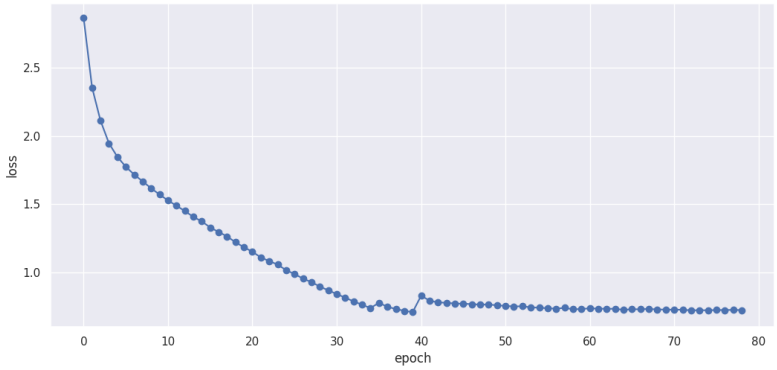


Figure 11: LSTM training loss curve.

verbs			nouns			actions		
top-1	top-3	top-5	top-1	top-3	top-5	top-1	top-3	top-5
19.22	44.0	55.99	3.80	10.98	17.74	0.71	2.27	4.36

Table 2: Acurracy of ConvNet1D on EPIC-KITCHENS-100.

on a restricted number of classes, we expect it to be even worse when more classes are used. But still, we observe the same phenomena as before, where the model seems to learn the task for a restricted number of classes, see Figure 12 (the confusion matrix can't be displayed as there are too many classes).

Unfortunately, there are currently no other baseline to compare with due to the recentness of the dataset.

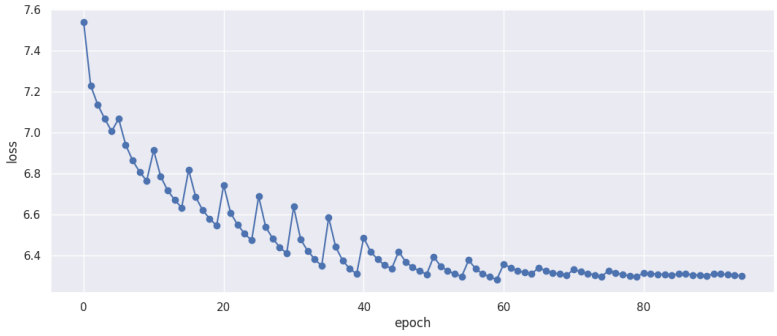


Figure 12: ConvNet1D training loss curve on EPIC-KITCHENS-100.

2.5 Tools

It's almost frightening the ease with which one can implement deep learning algorithms. Back in the 90's, researchers would take several weeks to train a simple ConvNet. Today, with python librairies, GPUs and storage devices that we have access to, such ConvNet can be train in hours only. The tools that we have in our disposition make deep learning looks like a children game, but can be dangerous if we use is without knowing what is under the hood. For this project we used the Keras API, that is built on Tensorflow (2.4), to develop, train and test our models, coupled with two NVIDIA GTX 1080Ti. We used distributed training to speed up the training because of the large size of one sample. Indeed, this large size prevented us from loading all the dataset in memory and training the models in one stage, we were therefore forced to resort to a workaround described in section 2.6.

2.6 Difficulties and problems

2.6.1 Memory

The major difficulty of this project was the memory management, for two reasons:

- Let's consider the size of one sample where the sequence length is fixed to $20 \rightarrow 20 * 256 * 256 * 3 = 3932160$. Each sample is a vector of almost 4 millions float32 values, which makes a total of 15Mb footprint for one such sample. Considering that we had 250Gb of RAM, it means that we can fit only 17K samples at a time, meanwhile our train set contained 67K samples.

- While a model was training using theoretically 30% of the RAM, because we divided the dataset in this way, we realized that it was actually using more than 60%. After some investigations we found that Keras was storing a duplicate of the data in the backend resulting in a surplus of memory usage, with even memory leak at some point.

These two reasons forced us to take memory management under serious considerations. We decided to divide the training procedure by training the models on groups of data. We divide our training/validation data in groups such that each group size is equal to a certain percentage of the RAM. We load the first group and train the model for a certain number of epochs and we record some validation metrics. We then do the same thing for each other

groups, at this point we go back to the frist group and iterate until all groups have been used n times; where n is the number of epochs. This lead to a huge waste of time of data loading into the memory, but is unfortunately necessary. However, a good practice that can saves time and memory space , is the data type used to store the data. When working with images, two number formats are ubiquitous, *unsigned integer* and *single-precision floating-point*, respectively denoted as $uint8$ and $float32$. These two formats encode data using 1 and 4 bytes respectively. $uint8$ can encode natural numbers between 0 and 255, which are the values a pixel can takes, it is well-suited for this reason. On the other side, $float32$ can encode decimal values in a really wide range. The advantage of working with images, is that we can store them in memory as natural numbers, and convert them to decimal value before using a batch; because deep learning models work with normalized decimal values. Thus, the data was stored and loaded as $uint8$, and when a batch is presented we convert it to $float32$ and normalize the values. We provide in appendix D a slightly detailed comparison of these two types.

2.6.2 Time

When training deep architectures with big datasets (or big samples in our case), the time needed fo a model to train can rapidly explode. This is the first problem that required our attention and that we had to work on. A first solution was to train our models with the process described above. Next, we could use more memory and use bigger groups and reduce the number of loading required for one epoch. We had in our disposition two GPUs, and most of the time we ran one model per GPU, so the memory had to be divided by two. Nonetheless, we had recourse to distributed training when a good set of hyperparameters was found, as well as for the last convNet1D model. It is worth noting that the last model trained on EPIC-KITCHENS-100 took more than six days for 100 epochs, even with two GPUs and 165Gb of RAM.

3. Conclusion

3.1 Results synthesis

Lorem ipsum.

3.2 Roads for investigation

Lorem ipsum.

3.3 Acquired knowledge and skills

Through this project I had the opportunity to work on a topic that is currently on the deep learning and computer vision research front scene. This project shown me the difficulty to deal with deep architectures, especially debugging such black boxes.

Bibliography

[Bertschinger and Natschläger, 2004] Bertschinger, N. and Natschläger, T. (2004). Real-time computation at the edge of chaos in recurrent neural networks. *Neural Computation*, 16(7):1413–1436.

[Damen et al., 2018] Damen, D., Doughty, H., Farinella, G. M., Fidler, S., Furnari, A., Kazakos, E., Moltisanti, D., Munro, J., Perrett, T., Price, W., et al. (2018). Scaling egocentric vision: The epic-kitchens dataset. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 720–736.

[Damen et al., 2020] Damen, D., Doughty, H., Farinella, G. M., Furnari, A., Kazakos, E., Ma, J., Moltisanti, D., Munro, J., Perrett, T., Price, W., et al. (2020). Rescaling egocentric vision. *arXiv preprint arXiv:2006.13256*.

[Erhan et al., 2009] Erhan, D., Bengio, Y., Courville, A., and Vincent, P. (2009). Visualizing higher-layer features of a deep network. *University of Montreal*, 1341(3):1.

[Farnebäck, 2003] Farnebäck, G. (2003). Two-frame motion estimation based on polynomial expansion. In *Scandinavian conference on Image analysis*, pages 363–370. Springer.

[Fukushima, 1981] Fukushima, K. (1981). Neocognitron–a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *NHK* , pages p106–115.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Hur and Roth, 2020] Hur, J. and Roth, S. (2020). Optical flow estimation in the deep learning age. In *Modelling Human Motion*, pages 119–140. Springer.

[Ji et al., 2012] Ji, S., Xu, W., Yang, M., and Yu, K. (2012). 3D convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231.

[Laurent and von Brecht, 2016] Laurent, T. and von Brecht, J. (2016). A recurrent neural network without chaos.

[LeCun et al., 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Miao et al., 2020] Miao, K.-c., Han, T.-t., Yao, Y.-q., Lu, H., Chen, P., Wang, B., and Zhang, J. (2020). Application of lstm for short term fog forecasting based on meteorological elements. *Neurocomputing*, 408:285–291.

[Olah et al., 2017] Olah, C., Mordvintsev, A., and Schubert, L. (2017). Feature visualization. *Distill*, 2(11):e7.

[Pérez et al., 2013] Pérez, J. S., Meinhardt-Llopis, E., and Facciolo, G. (2013). TV-L1 optical flow estimation. *Image Processing On Line*, 2013:137–150.

[Sandler et al., 2019] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2019). Mobilenetv2: Inverted residuals and linear bottlenecks.

[Shen et al., 2018] Shen, J., Pang, R., Weiss, R. J., Schuster, M., Jaitly, N., Yang, Z., Chen, Z., Zhang, Y., Wang, Y., Skerrv-Ryan, R., et al. (2018). Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4779–4783. IEEE.

[Shi et al., 2015] Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., and Woo, W.-c. (2015). Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *arXiv preprint arXiv:1506.04214*.

[Sudhakaran et al., 2019] Sudhakaran, S., Escalera, S., and Lanz, O. (2019). LSTA: Long short-term attention for egocentric action recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9954–9963.

[Wang et al., 2016] Wang, L., Xiong, Y., Wang, Z., Qiao, Y., Lin, D., Tang, X., and Van Gool, L. (2016). Temporal segment networks: Towards good practices for deep action recognition. In *European conference on computer vision*, pages 20–36. Springer.

[Wu et al., 2016] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

[Xu et al., 2015] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR.

[Ye et al., 2019] Ye, W., Cheng, J., Yang, F., and Xu, Y. (2019). Two-stream convolutional network for improving activity recognition using convolutional long short-term memory networks. *IEEE Access*, 7:67772–67780.

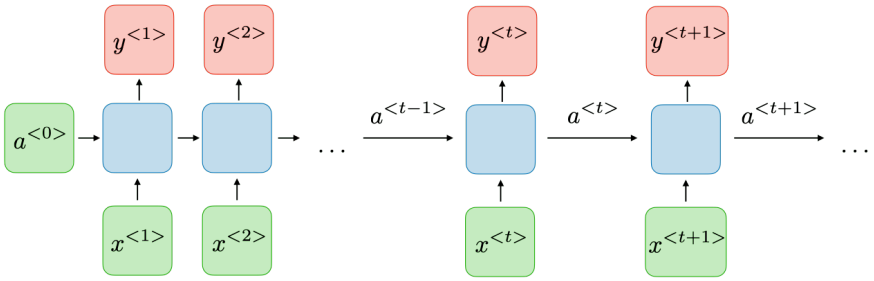
A. ImageNet and EPIC-KITCHENS comparisons

(a) EPIC-KITCHENS

(b) ImageNet

B. Reccurent Neural Networks / Long-Short Term Memory

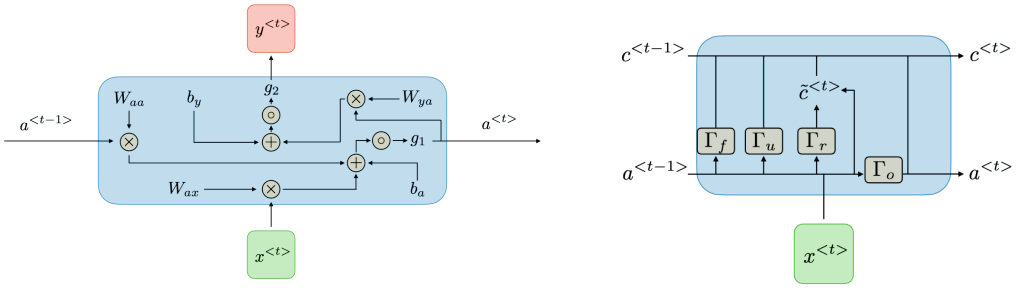
Processing time-sequence data with deep architectures has been challenging due to the necessity of incorporating a memory mechanism. To handle this, Recurrent Neural Network (RNN) use a recursive loop over the sequence elements while maintaining a memory state called *hidden state* reffered as $a^{<>}$ in Figure 13. This hidden state is charged to keep information flowing through the network by being updated at every step. A RNN cell, depicted in



https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Figure 13: vanilla RNN architecture.

Figure 14a, uses *gates* to encode, update and reset information; the learnable weights are $(W_{aa}, W_{ax}, W_{ya}, b_a, b_y)$.



(a) vanilla RNN cell. (b) LSTM cell.

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Figure 14: Recurrent cells description.

A successful implementation of RNN is called Long Short-Term memory [Hochreiter and Schmidhuber, 1997]. Its particularity is that it implements not one but two states called *hidden* (a_t) and *cell* (c_t) states. The purpose of the cell state in LSTM is to allow for long-term dependencies, inducing a long-term memory. While their memory cell are more complex than RNN's, see Figure 14b, they exhibit most of the time better performance due to their ability to catch these long-term dependencies. The flow of information through the cell state, top line of Figure 14b, has been shown to help counter gradient vanishing.

LSTM equations:

$$
\begin{aligned}
\Gamma_f &= \sigma(W_f x_t + U_f a_{t-1} + b_f) \\
\Gamma_u &= \sigma(W_u x_t + U_u a_{t-1} + b_u) \\
\Gamma_r &= \sigma(W_r x_t + U_r a_{t-1} + b_r) \\
\Gamma_o &= \sigma(W_o x_t + U_o a_{t-1} + b_o) \\
\tilde{c}_t &= tanh(W_c[\Gamma_r \circ a_{t-1}, x_t] + b_c) \\
c_t &= \Gamma_f \circ c_{t-1} + \Gamma_u \circ \tilde{c}_t \\
a_t &= \Gamma_o \circ tanh(c_t)
\end{aligned}
$$

where σ is the sigmoid activation function, $tanh$ the hyperbolic tangent function, $\circ$ denotes the Hadamard or element-wise product and the learnable weights (and biases) are W, U and b. (Γ_f, Γ_u, Γ_r, Γ_o) are the LSTM gates and their purpose is to forget, update, reset and reveal the information respectively. At the first step, c_0 and a_0 are set to 0.

C. Optical Flow fields

We provide some examples of visualization of optical flow between two RGB images (I_1 and I_2). For this we have to understand that (dense) optical flows are per-pixel motion estimation, which result to an horizontal and vertical motions. Thus, they are computed as an horizontal, called u, and vertical, called v, displacements. To combine these two, we can computes the magnitude of the optical flow field, which is $m = \sqrt{u^2 + v^2}$. All of these are shown in Figure 15.

(a) I_1

(b) I_2

(c) u

(d) v

(e) m

Figure 15: Optical flow field example.

D. Data type

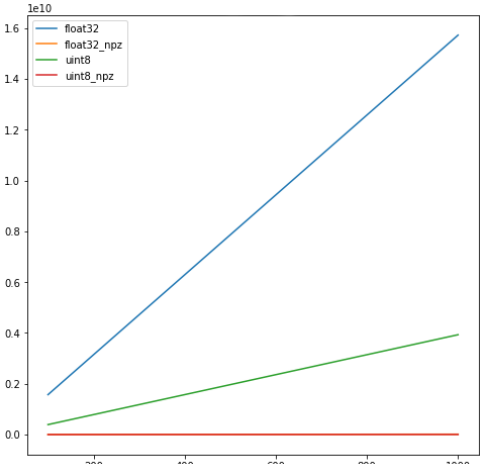
We didn't include implementation details in this report except for this to facilitate the comprehension of the discussed problem.

Managing memory carefully can benefit a considerable reduce of computation time and memory consumption. We show in Figure 16 different keypoint values with respect to the size of the dataset. To better understand what is shown here, we specify that these types are the types of the *numpy* arrays in which the data was stored.

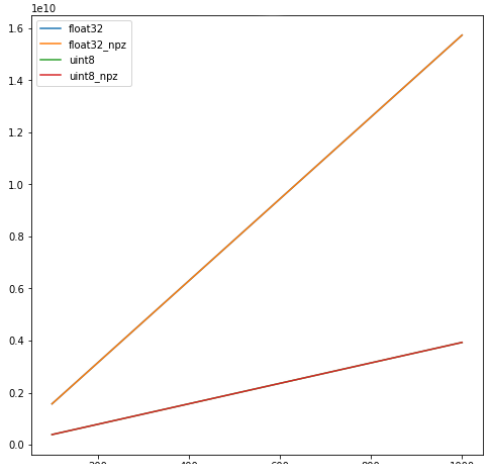
To save some space in disk, we also considered using a compressed version of the dataset that we compare along with the uncompressed version.
The legends depicted in the figures mean:

- $float32$ compressed $float32$
- $float32\_npz$ uncompressed $float32$
- $uint8$ compressed $uint8$
- $uint8\_npz$ uncompressed $uint8$

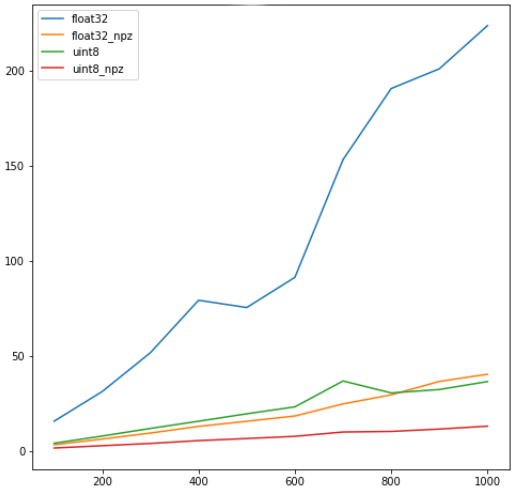
In 16a and 16b, we see the size, in bytes, that a dataset occupies in disk and memory with respect of N (number of samples); notice that for the memory, the compressed and uncompressed version are overlapping because compression is only taking place in disk. It show the size is linear with N, which is to be expected, and that the the higher N is, the bigger the difference between the two types is. In 16a, $float32\_npz$ and $uint8\_npz$ are overlapping and seem to be equal to zero, not because they are but rather because the sizes of the uncompressed versions are so big that they make the compressed version look so. Changing from *float32* to *uint8* divide the loading time (in seconds) by a factor of 2 or more (16c). However, using compressed data augment considerably the loading time, because of the decompression process (difference between the blue and the orange lines in 16c). Since, we load and train on groups of data rather than all the dataset at once, loading data is made several time during the training. If we consider using the compressed version, we choose to sacrifice time for disk space, and the solution is to find a good tradeoff. In my case, I was provided enough disk space to consider using uncompressed data, which saved me some time.



(a) disk size



(b) RAM size



(c) loading time

Figure 16: Comparison of $uint8$ and $float32$.

E. Confusion matrices

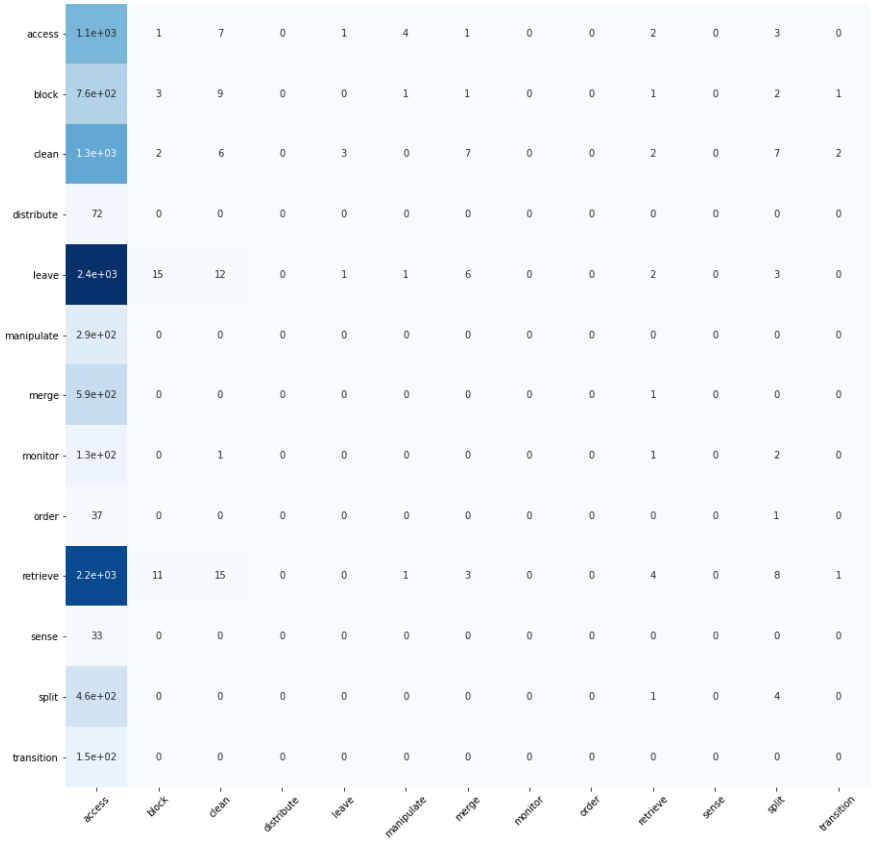


Figure 17: ConvLSTM confusion matrix.

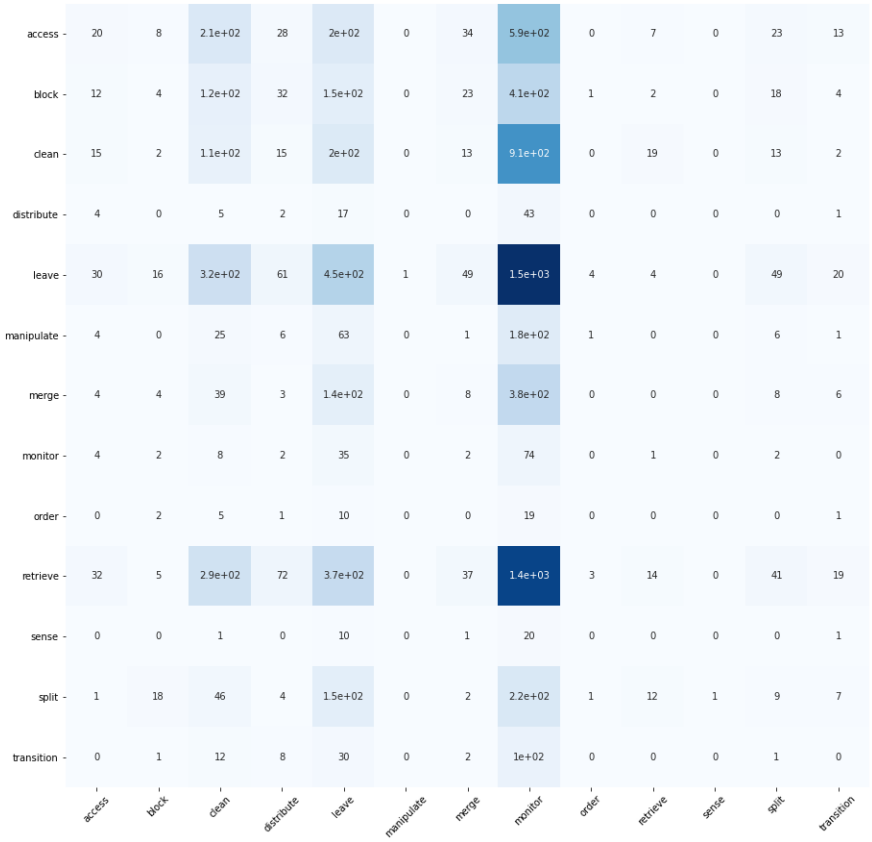


Figure 18: LSTM confusion matrix.

Keywords

Informatiques - Recherche - Algorithmes - Logiciel d'analyse de données - Deep Learning - Vision par ordinateur - Reconnaissance d'action - Vision égocentrique

GUETARNI Bilel **ST50 thesis - P2020**

Abstract

Le deep learning a été largement utilisé dans la vision par ordinateur durant les précédentes décennies. Plusieurs problèmes ont étés adressés notamment la détéction d'objets, la génération de contenu, la segmentation d'image ou d'instance et une poignée d'autres. De nos jours, des tâches très complexes sont au coeur de la recherche académique, incluant la reconnaissance d'action ou d'activité; la différence entre ces deux tâches est très fine et nous nous concentrons ici sur la reconnaissance d'action (la différence peut être vue comme par example l'action 'prendre une fourchette' et l'activité 'cuisiner').
Atteindre des résultats resonnables dans la reconnaissance d'action pourrait permettre de développer des systèmes de suivi de patient atteint de troubles physiques, comme des tremblements, dans leur vie de tous les jours. Il serait ainsi plus facile d'identifier les actions dont une personne éprouve du mal à effectuer, et ainsi conduire un processus de réhabilitation ciblée. Dans ce projet nous explorons comment le deep learning peut-être appliqué à la reconnaissance d'action et quel en est l'état de l'art actuel. Nous nous sommes concentré sur des solutions peu coûteuses en calcul puisque nous nous attendions à les implémenter sur des systèmes embarqués, e.g. téléphone mobile, NVIDIA Jetson Nano...

Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud
Avenue des Sports 29
1400 Yverdon-les-Bains
heig-vd.ch

utbm
université de technologie
Belfort-Montbéliard