Solving a classification task with the implementation of a *self-supervised* learning problem by solving the Jigsaw puzzle reassembly task

Dr. Elena Mattiazzo
Dr. Giovanni Cavallin
Dip. di Matematica
Università degli Studi di Padova
Email:
elena.mattiazzo.1@studenti.unipd.it
giovanni.cavallin.1@studenti.unipd.it
Matricola: 1206695, 1206693

***Abstract*—In this report we study and implement the problem of image representation learning without human annotation presented in [1] by Noroozi and Favaro. By following the principles of self-supervision, we build a convolutional neural network that can be trained to solve Jigsaw puzzles as a *pretext* task, which requires no manual labeling, and then later repurposed to solve object classification. To maintain the compatibility across tasks we implemented the *context-free network*, the siamese-eenead CNN presented in the paper. The CFN takes image tiles as input and explicitly limits the receptive context of its early processing units to one tile at a time. Our implementation shows that the learned featured by the pretext task can be successfully transferred to the classification problem of Food Images [2] through transfer learning.**

I. Introduction

Visual tasks, such as object classification and detection, have been successfully approached through the supervised learning paradigm. However, since manually labelled data is costly and not scalable, unsupervised learning is gaining momentum.
Recently a new unsupervised learning paradigm is raising: *self-supervised* learning. The idea is to exploit different labeling that is freely available besides or within visual data, and to use them as intrinsic reward signals to learn general-purpose features. For example, [3] uses the relative spatial co-location of patches in images as a label. In [4] they use object correspondence obtained through tracking in videos, and [5] uses ego-motion information obtained by a mobile agent such as the Google car [6]. The features obtained with these approaches have been successfully transferred to classification and detections tasks with encouraging performances when compared to the supervised task.
A fundamental difference between [3] and [4] is that the former method uses single images, while the latter use multiple images related through a temporal or viewpoint transformation. While it is true that biological agents typically make use of multiple images and other information, it is also true that single snapshot may carry more information than the one that has been extracted so far. In [1] Noorozi and Favaro worked on a novel supervised task the *Jigsaw puzzle reassembly* problem, which builds features that yield high performance when transferred to detection and classification task. The experimental evaluations show them that the learned features capture semantically relevant content, and their method outperforms state of the art methods in several transfer learning benchmarks.
In this report we would like to implement the method found in [1] and to apply the fine-tuning task to solve the classification task of Food Images [2] competition yielded by Kaggle [7].

II. Related work

The work that we are going to study is a problem of *representation/feature learning*, which is an unsupervised learning task. Representation learning regards the building of intermediate representations of data useful to solve machine learning tasks. In this project, the features that have been learned by solving the Jigsaw puzzle are repurposed to solve a classification problem. This involves also *transfer learning* via *fine tuning*, which is a process of updating the weights that are obtained from the training from a task to solve another task.

A. Unsupervised Learning

Most techniques that regards unsupervised learning exploit general-purpose priors such as smoothness, sharing of factors, hierarchical factors, etc. However, a general criterion to design a visual representation is not available. In the context of Computer Vision there are early works on unsupervised learning of models for classification. For example, in [8] and [9] a probabilistic representation of objects as constellations of parts is presented. A limitation is the high computational complexity of these models. We will explain later that the Jigsaw puzzle solver also aims to build a model of both appearance and configuration of the parts.

B. Self-supervised Learning

This kind of unsupervised learning exploits labeling that comes for "free" within the data. There are two types of labels that can be used:

- labels that are easily accessible and are associated with a non-visual signal (such as egomotion);
- labels that are obtained form the structure of the data.

The paper that we are going to implement uses the latter case as they simply re-use the input images and exploits the pixel arrangement as a label.
In [3] Doersch *et al.* train a convolutional network to classify the relative position between two patches. In contrast, in [1] the Jigsaw puzzle problem is solved by observing all the tiles at the same time. This allows the trained network to intersect all ambiguity sets and possibly reduce them to a singleton. Agrawal *et al.* [5] exploits labeling provided by an odometry sensor: they trained a Siamese network to estimate ego-motion from two images frames and compare it to the ego-motion measured with the odometry sensor. The advantage is that labeling is freely available in most cases. However, since the object identity is the same in both images, the intraclass variability may be limited: concentrating on the low-level similarities between the images, some high-level structures would be lost. The Jigsaw puzzle approach ignores the similarities between the tiles and instead focuses on their differences.

C. Jigsaw Puzzle

Solving Jigsaw puzzles has been associated with learning since their inception in 1760 by John Spilsbury. Studies in Psychonomic show that Jigsaw puzzles can be used to assess visuospatial processing in humans [10]. Instead of using Jigsaw puzzles with this purpose, in the referring paper they want to use them to develop a visuospatial representation of objects in the context of CNNs.

III. The Jigsaw Puzzle Problem

In the first part of this chapter we will discuss about the theory that has been used to set the problem. In the second part we will explain the implementation details that we adopted to implement and improve the performance of the paper.
An immediate approach to solve the Jigsaw puzzle is to stack the tiles of the puzzles along the axis of the color (so $9 \times 3 = 27$ channels). However, with this approach the net is caused to learn only low-level texture similarities instead of high-level primitives. We as humans use kind of analogies between the tiles - like similar patterns or contiguous borders between pieces - as cues to solve Jigsaw puzzles. However, this learning does not require any understanding of the global object. Thus, we need a network that delays the computation of statistics across different tiles and that can find the parts arrangement using these features. The objective is to force the network to understand representative features that can discriminate the relative location of each part of an object.

A. The Context-Free Architecture

The solution found in [1] is to build a siamese convolutional network (see fig. 1), where AlexNet (until layer *fc6*) is used for each crop of the puzzle, with shared weights. The outputs of all *fc6* layers are concatenated and given as input to the fully connected layer *fc7*. In this way each patch is given to a different AlexNet separately, until the very last fully connected layer: the *fc7* is responsible for the understanding of the rearrangement of the tiles - the context of the objet. Therefore this network architecture has been called *context-free*: each patch remains separated from the others until the last layer that handles the context of the object.
The performance of this architecture is similar to the AlexNet one in the classification task on the ImageNet 2012 dataset [11]. However, the CFN is more compact than AlexNet: the first depends on $27.5M$ parameters, the latter on $61M$. In addition, the network can be used interchangeably for different tasks including detection and classification. In the

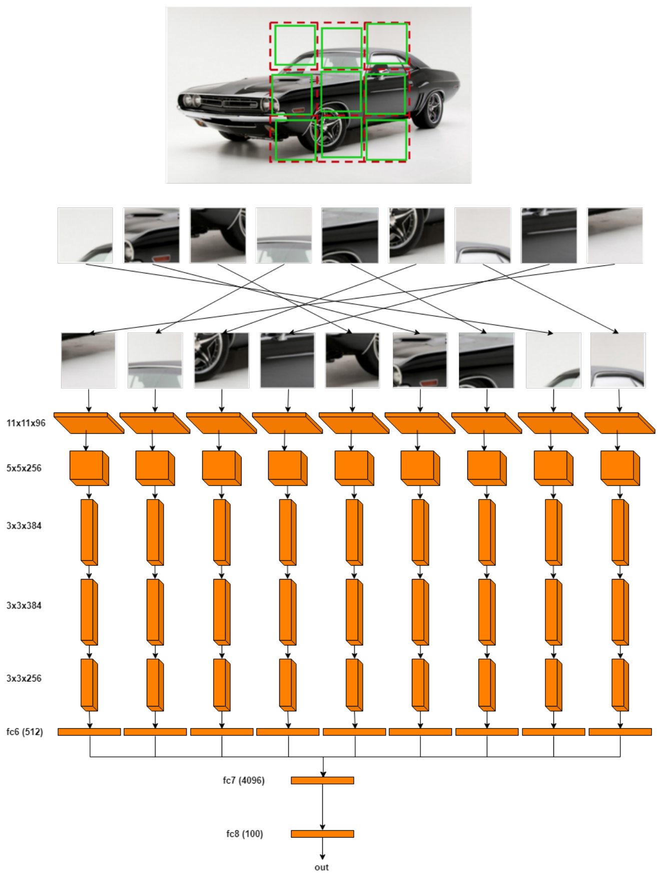

Fig. 1. The Siamese network. The layers inside the AlexNet block have shared weights. The AlexNet block is composed by the repeated layers.

next section we show how to train the CFN for the Jigsaw Puzzle reassembly.

B. The Jigsaw Puzzle Task

To train the CFN we define a set of Jigsaw puzzle permutation, *e.g.*, a tile configuration $S = (4, 2, 3, 5, 8, 7, 1, 0, 6)$. We

randomly pick one of those permutations, rearrange the 9 input patches that has been retrieved from an image accordingly to that permutation, and ask the CFN to return a vector with the probability value for each index. Given $9! = 362,880$ possible permutations, it is clear that this is an important factor on the performance of the representation that the network learns.
The permutation set controls the ambiguity of the task. If the permutations are too close to each other, the Jigsaw puzzle task is more challenging and ambiguous. For example, if the difference between two different permutations lies only in the position of two tiles and two of them are equal, the prediction of the right solution is impossible. To show this issue quantitatively, as specified in the ablation study of [1][4.2], the learned representation on the PASCAL VOC 2007 detection task has been compared using several permutation sets based on the following two criteria:

a) Cardinality: they trained the network with a different number of permutations to see what impact this had on the learned features. They found that as the total number of permutation increases, the training on the Jigsaw becomes more difficult. The performance of the detection task increases, though. As regard the experiments that we have made, we decided to use the first 40 permutation for the PC task and 100 for the GCP one, as the paper results were promising and the training has shown to be easier than the 1000 task. More information about those two tasks can be found at III-D1 and V.

b) Average Hamming Distance: it has been seen [1][p.11] that the average Hamming distance between permutations controls the difficulty of the Jigsaw puzzle reassembly task and is correlated with the object detection performance. As the average Hamming distance increases, the CFN gains higher accuracy and so does the fine-tuning tasks. For this reason we choose the higher possible Hamming distance. The algorithm that has been used is better described in Algorithm 1: it begins with an empty permutation set and at each iteration it selects the one that has the maximum Hamming distance to the current permutation set.

```
Input  : N number of permutations
Output: P maximal permutation set
1  // $\bar{P}$ is a $9 \times 9!$ matrix;
2  $\bar{P} \leftarrow$ all permutations $[\bar{P}_1, ..., \bar{P}_{9!}]$;
3  $P \leftarrow \emptyset$;
4  // uniform sample out of 9! permutations;
5  $j \sim U[1, 9!]$;
6  $i \leftarrow 1$;
7  while $i \leq N$ do
8      // add permutation $\bar{P}_j$ to $P$;
9      $P \leftarrow [P \ \bar{P}_j]$;
10     // remove $\bar{P}_j$ from $\bar{P}$;
11     $\bar{P} \leftarrow [\bar{P}_1, ..., \bar{P}_{j-1}, \bar{P}_{j+1}, ...]$;
12     // $D$ is a $i \times (9! - i)$ matrix);
13     $D \leftarrow$ Hamming(P, P');
14     // $\bar{D}$ is a $1 \times (9! - i)$ row vector;
15     $\bar{D} \leftarrow 1^T D$;
16     // $\bar{D}_k$ denotes the k-th entry of $\bar{D}$;
17     $j \leftarrow \arg\max_k \bar{D}_k$;
18     $i \leftarrow i + 1$;
19 end
```
Algorithm 1: Generation of the *maximal* Hamming distance permutation set

C. Training the CFN

The output of the CFN can be seen as the conditional probability density function (pdf) of the spatial arrangement of object parts (or scene parts) in part-based model, *i.e.*,

$$p(S|A_1, ..., A_9) = p(S|F_1, ..., F_9) \prod_{i=1}^{9} p(F_i|A_i)$$

where S is the configuration of the tiles, A_i is the i-th part appearance of the object, and $\{F_i\}_{i=1,...,9}$ form the intermediate feature representation. Our objective is to train the CFN so that the features F_i have semantic attributes that can identify relative position between parts. As the pdf is very high-dimensional, close attention must be paid to the training strategy. In general, a self-supervised learning system might lead to representation that are suitable to solve the pre-text task, but not the target task, *e.g.*, object classification, detection and segmentation. As regard this, an important factor to learn better representations is to prevent our model from taking undesiderable solutions, called solutions *shortcuts*. In [1] they have used multiple techniques that apply to different problems:

a) Absolute position: the CFN could learn to associate each appearance of A_i to an absolute position. In this case, the feature F_i carries no semantic meaning but just information about an arbitrary 2D position. To avoid this kind of learning we slowed down the training of the network in order to feed as much as possible Jigsaw puzzle per image; in addition, the maximum Hamming distance and the random selection of those patch reassembles assure us that different Jigsaw puzzle are chosen per each image with high probability. In this way the same tile should be assigned to multiple positions (ideally all 9) this making the mapping features F_i to any absolute position equally likely.

b) Edge continuity: to avoid shortcuts due to edge continuity and pixel intensity distribution we leave random gap between the tiles. In this way we discourage the network to learn context representation based on edges. During training the images are resized to a square of 256 pixels, preserving the original aspect ratio. Then they are cropped with a box of 225×225 randomly positioned and afterwards cropped into a 3×3 grid of 75×75 pixel tiles. From each of those tiles a 64×64 region is extracted randomly and then fed into the network. Thus, the average gap between each tile may range from a minimum of 0 pixels to a maximum of 22 pixels.

c) Chromatic aberration: chromatic aberration is a relative spatial shift between color channels that increases from the image center to the borders. This type of distortion helps the network to estimate the tile position. To avoid this kind of shortcut three techniques were applied:

- crop central square of original image randomly, getting a 225×225 pixels image;
- at training time about the 30% of the dataset is converted to grayscale
- spatial jitter over the color channels is applied randomly differently for each tile by ± 0, ± 1, ± 2 pixels.

As shown in [1][Tab.5], the application of all of these techniques makes the Jigsaw task more difficult - the accuracy is lower avoiding shortcuts - but the performance of the target task (detection in that case) is higher. So we decided to implement all of them.

D. The Dataset

The dataset for the Jigsaw puzzle solver task is composed by squared images. Every images must be cropped into pieces, then fed into the Siamese network. No labels required. This is why we decided to use the Object Localization dataset [12] of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC2017). This dataset contains 1.281M images for the training set, 50K validation images and 100K for the test. There are about 1K categories with a range from 732 to 1300 images each. All images are in JPEG format.

1) Select and crop images: once the images have been downloaded, we decided to build our own dataset on the basis of what we needed for the pre-task. So, we prepared two datasets, both divided into 70% training, 25% validation and 5% test:

- one made of a total of 500K random images, that served the model trained with a personal computer;
- one made of a total of 1.4M random images, that served the model trained with a virtual machine hosted in Google Cloud Platform.

More information about the computational power is provided in section V-D. For every image of both datasets we decided to:

- select only RGB images;
- take the shortest dimension to crop the image to a squared proportion. The crop position was random;
- resize the cropped image to (256×256) pixels with the Lancsoz algorithm;
- save the cropped resized image as new ones for further access.

Moreover we used the Welford's online algorithm [13] to calculate the mean and standard deviation of the training set, that was used later to make any scaled feature value the number of standard deviation away from the mean. This information, together with train, validation and test set dimensions has been stored inside a specific *h5* file.

2) The Hamming Set: to create the Hamming set the algorithm 1 has been implemented. In particular we used the maximum Euclidean distance between P and $\bar{P}$. As regard the network on the personal computer, we decided to generate 40 permutations; for the GCP one 100 as this was the best compromise [1][Tab.4] between the difficulty of the training of the pre-task and the results of the target task.

3) The Data Pipeline: for the implementation of the dataset we decided to use TensorFlow in Python. This was the best choice since we decided to use the same framework also for the design of the Siamese network (see III-E). We implemented the data stream trying to:

- reduce the memory usage of the dataset both for RAM and GPU;
- speed up the data flow from the persistency to the GPU.

We decided to use the TensorFlow *tf.data.Dataset* API [14]: this API provides many powerful tools to read images, manage the data stream and serve the dataset to the network. In a first moment we decided to store the whole dataset inside a *h5* format file, in which the images were store as *numpy* arrays. In this way the dataset was a big blob file, easy to manage and access. With this kind of persistency, we created a generator that read the images one at a time and serve to the *Dataset from generator* API of TensorFlow. This approach was successfull because there was a minimum memory usage since every image was loaded in a binary mode. While in the personal computer all the resources were used, once we moved to the Google Cloud Platform we realized that the online GPU was barely exploited. This was because the GPU was much more powerful, and it processed much more images than the generator could provide. So we decided to change approach: first we saved the images produced in III-D1 as new images on disk. Then we collected the all paths to the images, and let the *tf.io* API to read them. In this way the shuffling was applied to very little list of string object, and the TensorFlow API was responsible for reading and parsing every image in a concurrent way.
As regard the data transformations, we implemented every step in III-C. We implemented every transformation as a *lambda* python callable function, that we included in the *tf.data.Dataset* pipeline with the specific *dataset.map* method. This let TensorFlow to manage the reading and preprocessing of the images within the computational graph, in a concurrent (and distributed, if necessary) manner, mainly with the GPU cores. The steps are the following:

1) we created the list of paths to train/val/test images and a random list of permutation indexes of the same length of the list of paths; in addition, a pointer to the *h5* information file described above was provided. This file is useful to determine the dimension of the sets together with the mean and standard deviation of the training set at compile time, sparing computational power. The list of permutation indexes is generated every repetition of the dataset randomly, so the labels are always different for each epoch;
2) we provided the two lists to TensorFlow *tf.io* API, which is responsible for taking one element at a time. It also gives an "imperative" way-of-programming to implement the data pipeline;
3) now the image is a tensor of float array of numbers. From every image is subtract the training mean and it is divided for the training variance as specified in III-D1;

4) with a probability of 30% the image is then converted to grayscale within the training set;
5) the image is cropped as described in the section III-C0b: every operation (multiplication, addition, slicing, etc.) is made with the TensorFlow methods and types to keep the computation efficient;
6) every crop is jittered spatially as described in III-C0c;
7) the label is converted in the OneHot format;
8) the dataset is now served.

E. The Siamese Neural Network

The architecture of the Siamese Neural Network, within the AlexNet one, has been well defined in III-A and in [15]. Although the number of layers and the dimension of the kernels were strictly defined, we were quite free on the definition of the optimizers, the regularizes and the parameters. In addition, we also tried to insert some other layers - mainly batch normalization ones. In the next sections we will discuss about the framework that we used and the two planned networks

1) TensorFlow and the eager execution*:* TensorFlow's eager execution [16] is an imperative programming environment that evaluates operations immediately, without building graphs: operations return concrete values instead of constructing a computational graph to run later. This makes it easy to get started with TensorFlow and debug models, and it reduces boilerplate as well. Eager execution is a flexible machine learning platform for research and experimentation and provides:

- *an intuitive interface*: the programmer can structure the code naturally and use Python data structures;
- *easier debugging*: the operations can be called directly and be inspected by standard Python debugging tools;
- *natural control flow*: it can be used the Python control flow instead of the graph one, simplifying the specification of dynamic models.

We made the first implementation of the Jigsaw puzzle solver method with the computational graph. The debugging was painful; in addition, since the architecture is not a standard one, we encountered many troubles with the operations to be attached to the computational graph, such as the loss and accuracy ops. This is why we decided to move to this new framework and to implement the Jigsaw puzzle solver network from scratch.
Since we needed a Siamese network with shared weights, we could not take benefits from the *models* that *tf.keras* offers. So we implemented the pre-task neural network as a subclass of the *tf.keras.Model* API. This *Model* class exposes built-in training, evaluation and prediction loops; in addition, it gives the list of its inner layers and the saving and serialization APIs. In this way we wrote all the layers that we needed in the initialization of the class (the AlexNet's layers and the following ones). Then we set the behavior of the *call* function so:

1) the *AlexNet* block is called many times as the number of crops: the inputs are the different tiles, one per AlexNet;
2) the outputs (the *fc6* layers, one per each AlexNet of the block) are collected and concatenated;
3) the concatenation is then provided to the dense and then the output layer is computed.

This configuration let us save and restore the weights easily for the target task, since the architecture remains mostly the same, apart from the *call* method.

2) The PC architecture: We calls the architecture that we trained in the personal computer the "PC" architecture, in contrast with the "GCP" one that we trained in Google Cloud Platform.
The PC network is well represented in fig. 2. In this archi-

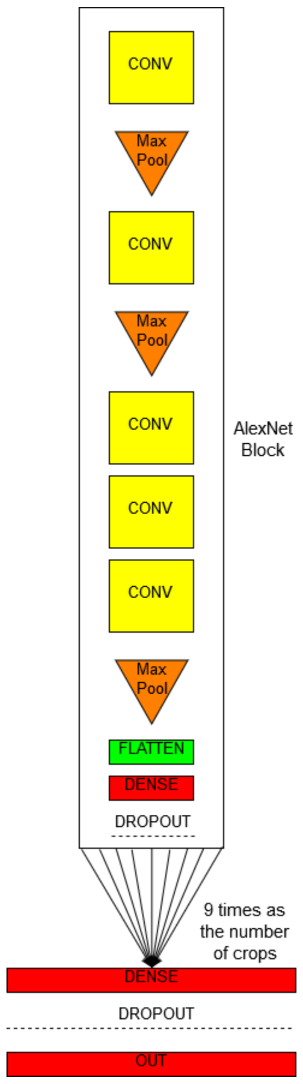

Fig. 2. Architecture of the PC neural network.

tecture we implemented all the features that were present in the original paper [1] with no additions. All the activations are *relu*s, the dimensions of the kernels are already listed in fig. 1. We used the Nesterov Stochastic Gradient Descent optimizer with momentum as precised in the paper. More information about the parameters and the training tactics are presented in V-A1.

3) The GCP architecture: The GCP architecture is showed in fig. 3. The main two changes are the batch normalization layers and the optimizer as we used Adam. The training task

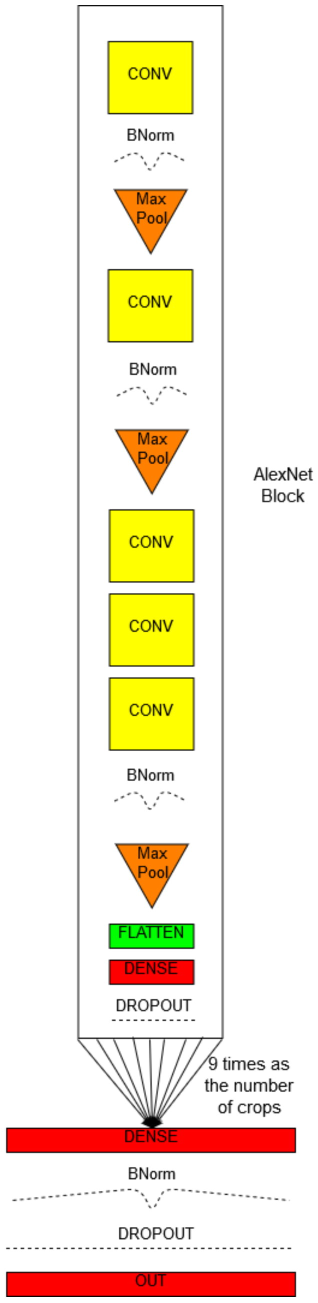

Fig. 3. Architecture of the GCP neural network.

we asked the network was the same as in the original paper, with slightly the same number of images (1.4M instead of 1.3M). So we decided to try to test different architecture to see if we were able to gain performances.
We decided to give batch normalization a chance since it has been demonstrated [17] that the reduction of the covariate shift between internal layers can speed up the training. The pre-task training was already faster than some other previous works [1][tab.1]; however, they decided not to use batch normalization layers. We were curious to see if this kind of normalization between convolutions with shared weights would have helped the training task.
As regard the optimizer, Adam is an adaptive learning rate optimization algorithm that has been designed specifically for training deep neural networks [18]. The paper contained some very promising diagrams, showing huge performance gains in term of speed of training. However, it has been studied that adaptive methods (such as Adam or Adadelta) do not generalize as well as SGD with momentum [19] when tested on a diverse set of deep learning tasks. Despite this we wanted to try and see the performance of this optimizer applied to our pre-task problem. More information about the parameters and the training tactics and results are presented in V-C1.

IV. Transfer Learning

In the experiments part of the paper was describe a practical use of the purpose of the project with transfer learning.

The formal definition of transfer learning is: "Given a source domain $\mathcal{D}_S$, a corresponding source task $\mathcal{T}_S$, as well a target domain $\mathcal{D}_T$ and a target task $\mathcal{T}_T$, the objective of transfer learning now is to enable us to learn the target conditional probability distribution $\mathcal{P}(\mathcal{Y}_T|\mathcal{X}_T)$ in $\mathcal{D}_T$ with the information gained from $\mathcal{D}_S$ to $\mathcal{T}_S$ where $\mathcal{D}_S \neq D_T$ or $\mathcal{T}_S \neq T_T$ "

So, transfer learning is a machine learning technique where a model trained on one task is re-purposed on a second related task. In the paper, the second related task is classification object and we chose to keep it.

The neural network for this part is the same of the first part, but we add an additional dense layer at the end, as in the paper.

We use the CFN weights that we have found at the end of the training of the CFN to initialize the `conv` layers of the standard AlexNet network. The other layers have Gaussian noise as initial weights. Then, those the `conv` layers are frozen during the process of training. In this way, this process allows rapid progress and improved performance when modelling the second task.

The transfer learning experiment that is describe in the paper use a dataset named "PASCAL VOC 2007". This dataset has 20 classes and $9,963$ images. We chose to use a different dataset to compare the results at the end of the training.

We select a dataset named "Food 101" [2] that have 100 classes and 1000 of images for each class. We choose to use a *.h5* file for it but to read the images from files. We provide to build a list of relative path to every image of the dataset with the corresponding associate class. Every class is traduce to an integer number.

The dataset was divided into 70% training, 25% validation and 5% test. We edit the images of the dataset in this way:

- central squared crop of every image;
- resize at 225×225 pixel;
- random flip up/down and left/right;
- random contrast between 3 and 10.

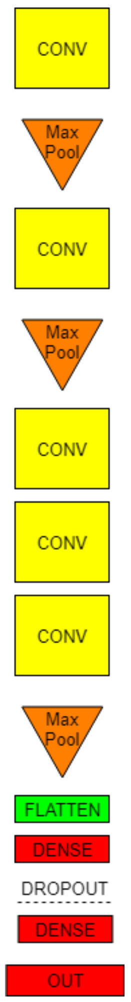

Fig. 4. Architecture of the transfer learning neural network.

V. Experiments

We trained the network for about two weeks, looking for the perfect set of parameters. However, two days before delivery we found a bug that invalidated all our results. We were instantiating the training, validation and test set with the same python method with a parameter, that indicated what kind of dataset we needed. The training and validation accuracy were coherent and was going up and down as we changed the hyperparameters, but the fine-tuning task was not giving good results. In addition, the evaluation of the network was random: once it was coherent with the training and validation accuracy, another time was completely different. We debug for many days until we found that probably the computational graph wasn't built too well in case of shared weights (we had not found any literature about this). So, we created three distinct (and mostly equal) functions that retrieve the training, validation and test sets: this structure let us train and evaluate the network. However, the time was too little to train the network profitably.
The experimental part is made up of two different parts even if the fine-tuning part depends mostly on the results of the pretext task.

A. Jigsaw puzzle task performance

As seen in III-E2 and III-E3 we used two different network architectures to train the Jigsaw puzzle that served two different purposes:

- the PC network to implement the paper pretext task *as-it-is*;
- the GCP network to try to outperform previous results in performances and execution time.

The training tactics have been slightly different, so they will be presented in two separate sections.

1) The PC network training: We trained the PC network making use of those assumptions:

- **tunable hyperparameters**: the tunable parameters were the batch size, the learning rate, its decay rate and the momentum of SGD optimizer;
- **hyperparameters search**: since the GPU RAM was not so big we set the batch size as high as possible to fit the entire GPU. In this way we set it to 100 both for training and validation. As regard the learning rate, its decay rate and the momentum we tried different combination until we got a good result. We read many articles to understand the best way to tune those parameters until one [20] gave us the theory and the *rule-of-thumb*: the more the momentum, the less the learning rate;
- **stopping policy**: we used the early stopping policy to stop the training whenever the validation loss started to increase. In this way we let the personal computer to run the session only until it was useful and to stop as soon as possible.

The best performance that we gained is: We can see from

Type	Value
Training accuracy	0.457
Training loss	2.081
Validation accuracy	0.454
Validation loss	1.915
Training time	3h33min
Learning rate	1e-3
Learning rate decay	0.97
SGD Momentum	0

TABLE I
Results for PC network

fig. 5 and 6 that the results are promising, although the network started overfitting as soon as it reached the 0.45% accuracy on validation set.

2) The GCP network training: The training of the GCP network has been more difficult since we were free to add and remove some other layers in the network. The best configuration we found before the bug was the one explained in III-E3 and gave very good results. However, we could not trust those results and we started the training from scratch. We found that the Adam optimizer was more difficult to train since its trend is more sensible to the learning rate and

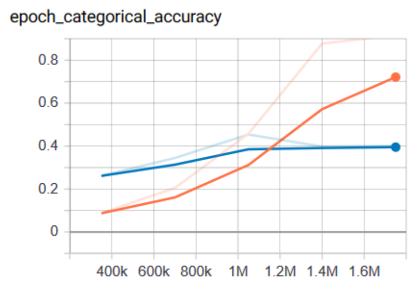

Fig. 5. Accuracy results of PC network

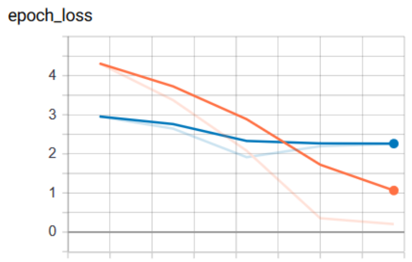

Fig. 6. Loss results of PC network

tend to overfit easily. We controlled that overfit with many batch normalization layers, that stabilized the weights between convolutions and before the output layers. With this in mind, we used those assumption to train the network:

- **tunable hyperparameters**: the tunable parameters were the batch size, the learning rate, its decay rate and the momentum of the batch normalization;
- **position of batch norm layers**: we tried different combination to see in which position the batch normalization layers were more effective in stabilizing the training. We have seen so far that they help more before the max pool layers of the AlexNet block and before the last dropout, near the output;
- **hyperparameters search**: to speed up learning without loss in generalization we set the batch size to 512, as high as we could. As regard the learning rate, we use it to slow down learning as much as possible, since the Adam optimizer had the tendency to converge rapidly (and to overfit). We tuned the learning rate decay with the same purpose. The batch normalization momentum was the trickiest hyperparameter to be tuned: too high carried no learning, too low overfit. We took inspiration from various online articles (see for example [21]), however none of them helped us very much. We finally found that *very* high momentum values were helping us to learn without overfit. In addition, we added a L2 regularization for weights and bias of the last AlexNet dense layer: this helped us to slow down learning and stabilize the loss function [22]. Therefore, the loss values are higher than the PC counterpart;
- **stopping policy**: as the online virtual instance of the machine is paid per hour and not for use we only set a maximum value of epochs. In case of overfitting we stopped the training as soon as we saw it.

The best performance that we gained is: We can see from fig.

Type	Value
Training accuracy	0.398
Training loss	4.898
Validation accuracy	0.367
Validation loss	5.202
Training time	6h6min
Learning rate	5e-6
Learning rate decay	0.80
BN Momentum	0.99 AlexNet, 0.997 last

TABLE II
RESULTS FOR GCP NETWORK

7 and 8 that the results are promising, although the network started overfitting as soon as it reached the 0.36% accuracy on validation set.

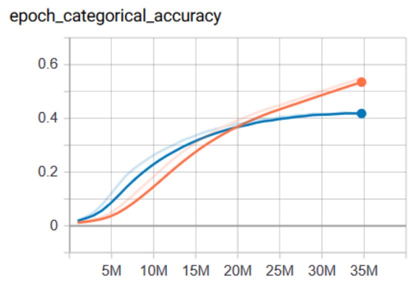

Fig. 7. Early accuracy results of GCP network

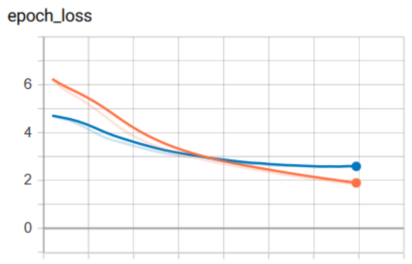

Fig. 8. Early loss results of GCP network

B. Fine-tuning on Food Dataset

As we have already explained, the little time we had to train the JPS network - and its consequently poor performance - did not let us have great results also in the fine-tuning task. Indeed, the results are really bad.
We will discuss in two different sections the results that we have obtained from the weights of the PC and the GCP networks. We have trained this network only with the personal computer since the free credits of GCP were exhausted.

1) Assumptions of the training: We trained the target task making use of those assumptions:

- **tunable hyperparameters**: the tunable parameters were the batch size and the learning rate;

- **hyperparameters search**: the batch size and the learning rate are bounded: the more the first, the more the latter. However the batch size that was fitting the personal computer was not so much, so we kept the learning rate very low;
- **stopping policy**: we used the early stopping policy to stop the training whenever the validation loss started to increase. In this way we let the personal computer to run the session only until it was useful and to stop as soon as possible.

C. *Performance coming from the PC network*

The best performance that we gained is: We can see from

Type	Value
Training accuracy	0.0221
Training loss	5.087
Validation accuracy	0.014
Validation loss	5.383
Training time	30min
Learning rate	1e-5
Learning rate decay	0.97
Batch size	50

TABLE III
RESULTS FOR THE TRANSFER LEARNING TASK COMING FROM THE PC NETWORK

fig. 9 and 10 that even if the time spent for the training is little, the results are far from promising.

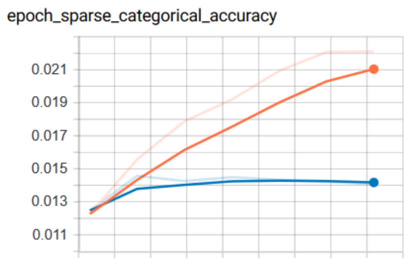

Fig. 9. Accuracy results of the transfer learning task coming from the PC network

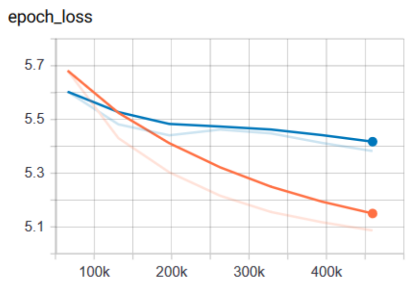

Fig. 10. Loss results of the transfer learning task coming from the PC network

1) Performance coming from the GCP network: The best performance that we gained is: We can see from fig. 11 and 12 that even if the results are better than the one that comes from the PC network, they are far from satisfying.

Type	Value
Training accuracy	0.0423
Training loss	5.083
Validation accuracy	0.025
Validation loss	6.789
Training time	23min
Learning rate	1e-6
Learning rate decay	0.97
Batch size	50

TABLE IV
RESULTS FOR GCP NETWORK

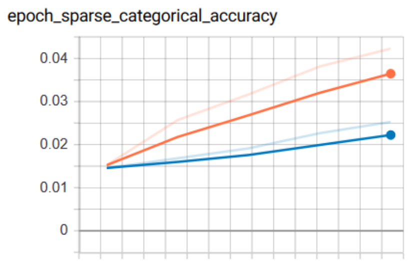

Fig. 11. Accuracy results of the transfer learning task coming from the GCP network

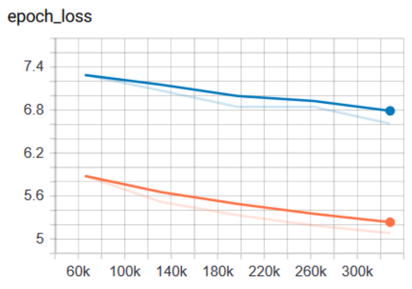

Fig. 12. Accuracy results of the transfer learning task coming from the GCP network

D. *Machines details*

The personal computer with which we have trained the pre-task neural network is a laptop with:

- CPU: Intel Core i5 8300H (2.30GHz, 3.90GHz Turbo);
- GPU: NVidia GTX 1050 (4GB RAM);
- Memory: 16GB DDR4 2666MHz;
- Persistency: SSD Samsung 970 evo NVMe;
- O.S.: Windows 10 Home.

The virtual machine that we have instantiated is composed of:

- vCPU: 4-core;
- GPU: NVidia Tesla T4 (12GB RAM);
- vMemory: 7.8GB
- Persistency: SSD 65GB;
- O.S.: Ubuntu 18.10.

VI. CONCLUSION

We have introduced the *context-free* network (CFN), a CNN whose features can be easily transferred between detection and classification and Jigsaw puzzle reassembly task. The network is trained in an unsupervised manner by using the Jigsaw puzzle as a *pretext* task. The learned features are then used for the classification task of Food recognition competition yielded by Kaggle. We have seen that the training of the Jigsaw task is very difficult and that the implementation of non-trivial models is a bit tricky with the TensorFlow framework. The results showed in V ar far from the one that the authors of the original paper were able to obtain, mainly as regards the target task. Since little time has been given to the training of the fine tuning, the bad performance may not come unexpected, however we believe that the poor performance of the Jigsaw puzzle solver is the mean reason why also the target task performed so badly.

REFERENCES

[1] M. Noroozi and P. Favaro, "Unsupervised learning of visual representations by solving jigsaw puzzles," *Lecture Notes in Computer Science*, p. 69–84, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46466-4_5

[2] K. Mader, "Food images (food-101)," 2018-05-15. [Online]. Available: https://www.kaggle.com/kmader/food41

[3] C. Doersch, A. Gupta, and A. A. Efros, "Unsupervised visual representation learning by context prediction," *CoRR*, vol. abs/1505.05192, 2015. [Online]. Available: http://arxiv.org/abs/1505.05192

[4] M. Weber, M. Welling, and P. Perona, "Unsupervised learning of models for recognition," in *Computer Vision - ECCV 2000*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 18–32.

[5] P. Agrawal, J. Carreira, and J. Malik, "Learning to see by moving," *CoRR*, vol. abs/1505.01596, 2015. [Online]. Available: http://arxiv.org/abs/1505.01596

[6] D. M. Chen, G. Baatz, K. Köser, S. S. Tsai, R. Vedantham, T. Pylvänäinen, K. Roimela, X. Chen, J. Bach, M. Pollefeys, B. Girod, and R. Grzeszczuk, "City-scale landmark identification on mobile devices," in *CVPR 2011*, June 2011, pp. 737–744.

[7] K. Inc. (2019) Kaggle. [Online]. Available: https://www.kaggle.com/

[8] R. Fergus, P. Perona, and A. Zisserman, "Object class recognition by unsupervised scale-invariant learning," in *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, vol. 2, June 2003, pp. II–II.

[9] M. Weber, M. Welling, and P. Perona, "Unsupervised learning of models for recognition," vol. 1842, 06 2000, pp. 18–32.

[10] J. T E Richardson and T. Vecchi, "A jigsaw-puzzle imagery task for assessing active visuospatial processes in old and young people," *Behavior research methods, instruments, computers : a journal of the Psychonomic Society, Inc*, vol. 34, pp. 69–82, 03 2002.

[11] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009, pp. 248–255.

[12] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[13] Wikipedia. (2019) Algorithms for calculating variance. [Online]. Available: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm

[14] Tensorflow. (2019) tf.data.dataset: a tensorflow 1.14 api. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

[16] Tensorflow. (2019) Eager execution. [Online]. Available: https://www.tensorflow.org/guide/eager

[17] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[18] V. Bushaev. (2018) Adam — latest trends in deep learning optimization. [Online]. Available: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

[19] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The Marginal Value of Adaptive Gradient Methods in Machine Learning," *arXiv e-prints*, p. arXiv:1705.08292, May 2017.

[20] V. Bushaev. (2017) Stochastic gradient descent with momentum. [Online]. Available: https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d

[21] I. R. (2018) Batchnorm: Fine-tune your booster. [Online]. Available: https://medium.com/@ilango100/batchnorm-fine-tune-your-booster-bef9f9493e22

[22] S. Palachy. (2018) Understanding the scaling of l2 regularization in the context of neural networks. [Online]. Available: https://towardsdatascience.com/understanding-the-scaling-of-l%C2%B2-regularization-in-the-context-of-neural-networks-e3d25f8b50db

[23] Tensorflow. (2019) Writing layers and models with tensorflow keras. [Online]. Available: https://www.tensorflow.org/beta/guide/keras/custom_layers_and_models