Open-Set Domain Adaptation through Self-Supervision

Andrea Cominelli, Edward Manca, Chunbiao Huang
Politecnico di Torino
s291423@studenti.polito.it s292493@studenti.polito.it s275935@studenti.polito.it

Abstract

Today, Big Data computation requires more and more data, and, for some problems, labels are more a need than a request, while it's often considered a luxury. Self-supervised tasks can help researchers capture meaningful knowledge from one domain and apply it to others. In this research, we study the use of self-supervised tasks in open-set domain adaptation problems by first training a self-supervised classifier together with a supervised classification task to discriminate between known and unknown classes in the target domain.
The considered task is a simplified version of the Rotation-based Open Set (ROS) method [1]: it consists in applying the rotation self-supervised task in order to extract meaningful knowledge from unlabeled data. Furthermore, we tackled other type of tasks, such as Flip self-supervised task and Jigsaw self-supervised task.
Chapter 3 summarizes the implementation of the 3 tasks and it describes the architecture, chapter 4 contains a series of experiments conducted on the basic architecture and chapter 5 a quick overview of the performance of the 3 tasks for the multi-head architecture.
*Link to the Github repository. The correct branch is the one named "**multi-head-architecture**"*

1. Introduction

People around the world exchange millions of photos, messages and reviews through their favorite device, feeding algorithms and machines that can improve their everyday life: think about self-driving vehicles, voice assistants such as Amazon Alexa and advanced surgical operations aided by robots. These technologies are only a small portion of what humans can achieve with gigantic amounts of data at their disposal.
Unfortunately, in order to properly train such algorithms, ground truth labels are more and more required and it became very difficult to manually annotate pieces of data that every hour are shared and captured. It is also very easy to make mistakes during the labeling phase, compromising the correct training of the model. Having said that, this path is by no means viable.
One possible solution consists in using existing labeled datasets in order to acquire some knowledge that can be reused on new unlabeled datasets. By doing this, the model should learn significant features that can be used to infer on unlabeled closely-related data. This process is called Domain Adaptation: it consists of the ability to train a model over a labeled set called "source set" and then exploit the generated knowledge over a different but closely-related unlabeled set: the "target set". We refer to a "Closed-Set Domain Adaptation" problem if the target set contains many categories as the source one. On the contrary, in real-world scenarios, it is common to deal with "Open-Set Domain Adaptation" problems, because the target set often contains more categories than the source one. In this particular case, we first have to distinguish in the target set between categories that belong to the source set and the other ones. Then we can solve a "Closed-Set Domain Adaptation" problem between the source set and the known portion of the target set.
In this paper we describe how self-supervised tasks can be used in order to achieve good results [4]. To solve the first step, we conduct experiments over a simplified version of the ROS method developed by Bucci et al [1]. The goal of the model is to predict the rotation that has been applied to images belonging to the target set and then, based on the prediction score, considers such images as if they belonged to known categories or not.
Once we correctly separate the target set, we collect all the unknown samples under the "unknown" category and reuse the same self-supervised task we implement in step one to reduce the domain shift between one set and the other.
Experiments were conducted over OfficeHome dataset[2] and various self-supervised tasks have been investigated such as guessing the flip direction on a flipped image and predicting the correct permutation index on an image having the relative portions mixed by the aforementioned permutation. Section 3 describes all the details about the used self-supervised tasks and the architecture implementation. Section 4 contains studies about hyperparameters. Finally

we present a multi-head version of the proposed architecture, accurately described by [1], and show its result compared against those provided by the basic idea.

2. Related work

Anomaly detection is a task that can be used to discriminate samples between the unknown and the normal class. Various different approaches for anomaly detection have been used as applied to the open-set scenario. Golan and El-Yaniv [5] present a method for using geometric transformations to create a self-labeled dataset. In this way the model should learn features that are useful for the detection of anomalies.
Closed-set **domain adaptation** has been studied extensively in recent years [2] however for real-world applications the closed-set assumption may not be correct. Open-set problems are much more reliable, thus many more techniques were born in order to tackle this problem. Recent studies in this field include 2 and 4.
Self-supervised learning has played, in recent years, a key role in maximizing the usefulness of unlabeled data. It consists of choosing a self-supervised task (or pretext task) to train alongside the main classification task: in this way, the model should extract useful knowledge reusable in other jobs. One possible self-supervised task is guessing what index permutation has been applied onto the image patches. This process has been already discussed in Carlucci et al [6].

3. Method

3.1. Self-supervised tasks

3.1.1 Rotation

To solve the Open-set domain adaptation problem we implemented a simplified version of the ROS strategy.[1]
The first step is to train a model capable of discriminating objects belonging to the target set between known and unknown categories, depending on whether an object category also belongs to the source set or not.
To accomplish this task, our model, trained on the source set, must be able to recognize a domain-invariant image transformation: if the model guesses the self-transformation used on a target set object it means that probably the object category is known, otherwise, it will be labeled as unknown. The reason why this is working it's because the model must learn at the same time weights capable of recognizing objects and domain invariant knowledge [3]. The studied transformation involves a random clockwise rotation applied to each image on the source set resulting in 0, 90, 180, or 270 degrees, as shown in Figure 1.
Sometimes, the rotation itself is not enough since can be some ambiguity in predicting the absolute orientation. For this reason, our implementation consists in concatenating features of the original image with the ones produced by the transformed image forcing the classifier to learn both the actual shape of the object and its rotated form instead of learning about texture patterns. Figure 1 better shows what type of features the model will learn by implementing this strategy.

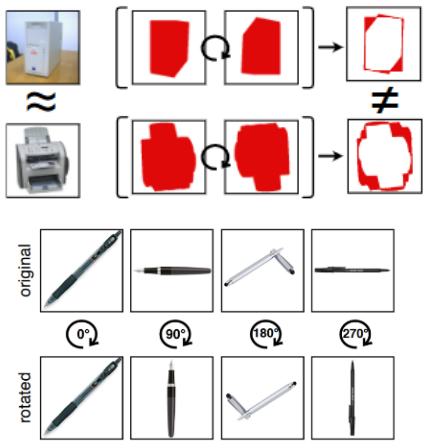

Figure 1. It is difficult to infer the absolute rotation without looking at the original object position (bottom picture). It is way simpler to predict such rotation by looking at the original position (top picture).

3.1.2 Flip

By performing this task, the original image is vertically flipped (Figure 2) and the network must guess if the image has been flipped or not. Also in this case, it would be pretty hard to guess which transformation has been applied without looking at the original image. Due to that, we again decided to keep concatenating the original feature map with the one computed by flipping the image.

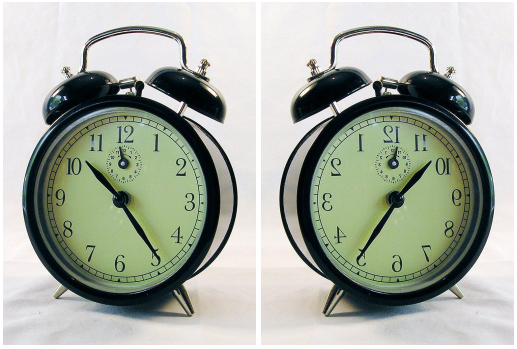

Figure 2. On the right the original image. On the left the flipped one

3.1.3 Jigsaw puzzle

A jigsaw puzzle is realized by extracting n patches from the original image and then shuffle them in order to compose another image, with all the patches concatenated in a random order (Figure 3). This task is particularly difficult because we force the model to infer which permutation has been applied to the original image. To make it easier we decided to select only jigsaw transformations for which the hamming distance between the permutations is maximum. The algorithm to identify which jigsaw transformations are the best ones is exactly the one present in [7].
During our experiments we divided the image in 3x3 patches and the number of permutations considered, according to the previous criterion, is 7: by changing this parameter, we found that 7 could be the perfect number for balancing the task difficulty and the processing time. Again, before forwarding the data to the self-supervised classifier, the original feature map is concatenated with the one computed from the transformed image.
Figure 3 shows an example of patches permutation.

3.2. Implementation

As already described, the proposed architecture must work on two steps: first, it must discriminate between known and unknown class samples on the source domain. Second, it must perform domain adaptation between the known part of the source domain and the target domain.
Both step 1 and step 2 are trained using a weighted

Figure 3. Jigsaw puzzle. The ground truth is represented by the index used to perform the permutation between all the patches i.e i = 1 perm = 1,2,3,4,5,6,7,8,9; i = 2 perm = 9,2,3,4,5,6,7,8,1 and so on. The numbers are referring to the index of the patches, starting from top left to bottom right.

combination of Cross Entropy loss. The hyperparameter study will be investigated later on.

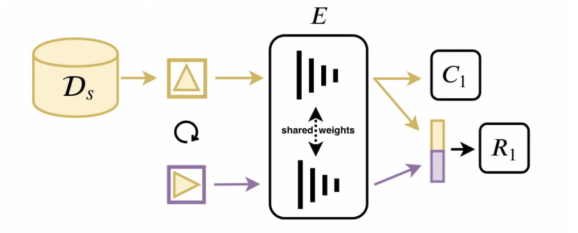

Figure 4. Step 1 architecture

3.2.1 Step 1

The first step is to implement a ResNet18 model, as done in [8], using it as backbone; in addition two heads, as MLP layers, are implemented. While the convolutional backbone is shared between the two heads, as it is common in multi-task learning scenarios, the two heads will be indipendent per each class. The first head has 45+1 output neurons (45 are one for each class of the source domain dataset, there is another neuron to identify the unknown class) while the second one has 4 (one for each possible rotation). An overview of step 1 final implementation is visible in Figure 4. The backbone layer will update its weights by optimizing a combined loss function

$$L_{tot} = L_{cls} + \alpha_1 L_{rot} \tag{1}$$

which puts together two cross entropy losses weighing them with the hyperparameter α_1.
As already mentioned, the input layer of the rotation head will be doubled in size since it expects both the feature map of the original image and the one produced by the rotation.

3.2.2 Evaluation

At evaluation time, the knowledge extracted by the model at the previous step will be applied to the target set. In order to establish if the image is coming from a known set or not, the network will compute its normality score which it will be compared with respect to a given threshold. The normality score is computed in the following way

$$NS(x_i)) = (\frac{1}{N}\sum_{j=1}^{N=4} max(R_1(E(x_i^j)))) \quad (2)$$

first we pass to the network each transformed image plus the original one and, for each of them, we extract the maximum score of the prediction. Finally we compute the mean of this list of predictions.
If the normality score is greater than a given threshold *t*, the sample will be considered as it is coming from a known class, unknown otherwise. During this phase the sample path, and its label, is saved to the known or unknown set, accordingly to the outcome of the operation. The new source set contains all source set samples plus target samples classified as unknown. In this way the final classifier will be able to model the unknown class Target samples classified as known will be stored in a different set. Training is performed in step 2. The overview for this step is visible in Figure 5.

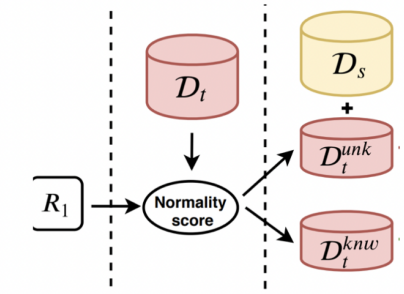

Figure 5. Evaluation step architecture. After this step, we get two new datasets that we use to perform domain adaptation on step 2. The problem is then reduced to a CSDA challenge.

3.2.3 Step 2

The final step consists in solving a closed set domain adaptation problem between the source set and the known part of the target set. Furthermore, we extend the model capabilities by defining the "unknown" class, which contains all the samples considered as unknown at evaluation time.
We use the same multi-task architecture we already used in step 1. The final classifier will, hopefully, be capable of extending its classification knowledge also on the unlabeled target set thanks to the self-supervised task we already discussed.
Again, the total loss is computed as

$$L_{tot} = L_{cls} + \alpha_2 L_{rot} \quad (3)$$

which is combining two cross entropy loss function, one for each head, by an hyperparameter α_2. The overview also in this case is visible in Figure 6

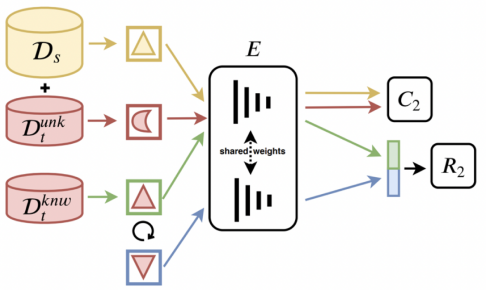

Figure 6. Step 2 architecture.

4. Standard architecture: experiments

Experiments were conducted over the Office-Home dataset [2]. It is composed by four different sets, each one with its own distribution: Art, Clipart, Product and Real world. Each of them is described by two different subdomains: the source and the target one. In order to properly discriminate between the two sets, the first 45 classes (picked up in alphabetical order) are considered the source domain and the target domain is built by adding 20 more categories to these ones.
As already mentioned, experiments were conducted on a ResNet18 [8]. We first analyze the base architecture and its performance, using learning rate 0.001 and weight decay 0.0005 with batch size of 128. Lastly, we look at the performance of the multi-head version of the proposed architecture with learning rate 0.003. Batch size and weight decay have not been modified. In both architectures, learning rate decay was turned off.
During this chapter we will present a complete hyperparameter ablation study performed on the Rotation self-supervised task (Flip and Jigsaw tasks will be analyzed as well but in a less deep manner, since the strategy is the same we adopted for the Rotation task) using the Art dataset as the source set and Clipart as the target one.
At training time, we apply some image augmentation, such as random crop, jitter and random grayscale. Random blur is implemented as well but has not been applied since it did not seem to improve the quality of our results.

AUROC score by α_1		
Epochs	AUROC	α_1
80	0.4829	0.1
	0.5105	**1**
	0.5053	2
	0.5092	4
	0.5084	8
	0.5051	16
50	0.4973	1

Table 1. AUROC scores organized by α_1 computed over 80 epochs. Second row shows the behaviour of the maximum AUROC when the number of epochs is reduced to 50.

4.1. Rotation self-supervised task

The Rotation self-supervised task is the first one that we analyze. It consists in predicting a random clockwise rotation applied to the image among four different rotations: 0, 90, 180 and 270 degrees. This type of transformation has been already discussed in detail by Gidaris et al [3].

4.1.1 Step 1: train the model on the source set

The first step consists in finding the optimal value for α_1 by training the network on the source set. The theoretical optimal value is the one that maximises the AUROC score, that is a particular metric capable of computing the precision of the normality scores over the target set at evaluation time.
α_1 describes how much the rotation classifier loss is impacting on the total loss. The higher the value is, the more it will influence the weight update during the backpropagation step.
Figure 7 and 8 show the accuracy trend respectively for the object classifier and the rotation classifier during 80 epochs for each value of α_1 chosen from 0.1 to 16. Table 1 summarizes the AUROC score for each value of α_1 for 80 epochs.
The best AUROC score has been registered around α_1 = 1. Furthermore, looking at the rotation classifier accuracy plot, the line corresponding to the theoretical best weight stops increasing fast around 50 epochs. By knowing this, we decided to perform an early stopping in order to speed up the experiment and, hopefully, avoid overfitting on the source set. On the other hand we caused a slightly drop of the found optimal AUROC score.

4.1.2 Threshold selection

Once we found the best theoretical α_1 it is time to search the best theoretical threshold *t* to perform the known-unknown separation of the target set: target samples with normality score higher than *t* will be considered as known, the others

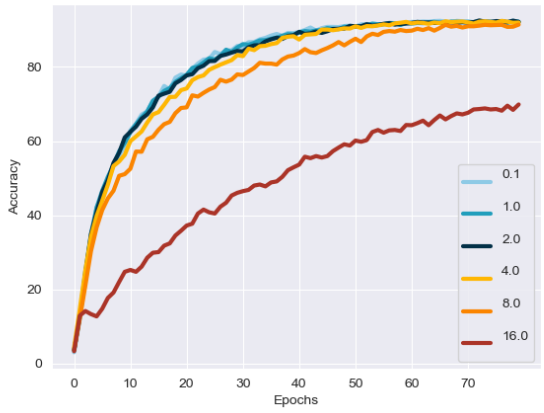

Figure 7. Object classifier accuracy measured over 80 epochs by 6 different values of α_1

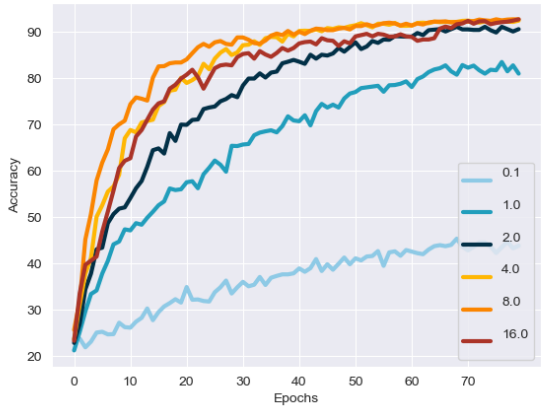

Figure 8. Rotation classifier accuracy measured over 80 epochs by 6 different values of α_1

will be considered as unknown. The target domain contains 3064 samples coming from the known domain. Having said that, in this implementation we consider the best threshold the one able to recognize a number of samples as belonging to the known set as close as possible to 3064, this choice is made for sake of simplicity, to not unbalance the dataset and to not risk having few samples over which the known/unknown separation can be performed.
Of course, this strategy, as said, is not the best one since we do not know if the recognized samples are actually belonging to that particular set or not. Moreover, in [1] the presented solution consists in setting the threshold as the arithmetic mean of the normality scores.
Table 2 summarizes the number of recognized samples classified by the proposed solution, organized by different thresholds. In order to find the value that is the closest pos-

Number of known samples by threshold t	
Known samples	t
4365	0.1
4365	0.3
3065	**0.5**
473	0.7
4	0.9

Table 2. Target samples recognized as known by different threshold values.

sible to the optimal one, we start searching among a fixed list of possible values (from 0.1 to 0.9 with 0.2 as step) and then, based on the results, we looked for intermediate values until we were satisfied with the obtained results.
In this case, threshold t = 0.5 is the best one: the number of samples recognized as known is very close to the ground truth quantity.

4.1.3 Step 2: Domain adaptation

Step after step, the scenario becomes closer and closer to the one of CSDA problem. Indeed, the final goal consists in performing domain adaptation between the source set and the known part of the target set. The proposed methodology is very similar to the one already described by Bucci et al [1]. Furthermore, by discriminating between known and unknown sets in the previous step, we now have the possibility to train the model to also recognize samples coming from the more generic unknown class.
To measure the goodness of the final step, we used three metrics, already exploited in [1]:**OS***, the accuracy of the object classifier in recognizing the known category; **UNK**, the accuracy of the object classifier in recognizing the unknown category; **HOS**, the harmonic mean between OS* and UNK.
The final step consists in tuning the hyperparameter α_2 whose task is to weight the rotation classifier loss when performing domain adaptation with regards to the total loss.
During this step we used the HOS metric to judge the performance of the model according to the chosen hyperparameter value: the highest it is, the more the model will be effective.
Table 3 contains all the metrics captured by the model summarized by α_2 values over 30 epochs of training. The range of possible α_2 values used in this step is the same we already discussed in step 1. It is worth mentioning for clarity that also in this case the original number of epochs has been reduced (from 50 to 30) because after more or less 30 epochs the HOS metric stops increasing.

Final metrics by α_2			
OS*	UNK	HOS	α_2
0.5094	0.7872	0.6185	0.1
0.5127	0.7679	0.614	1
0.5183	0.7786	**0.6223**	**2**
0.5089	0.7679	0.6122	4
0.4859	0.8064	0.6064	8
0.3681	0.7786	0.4999	16

Table 3. Target samples recognized as known by different threshold values. It seems that for α_2 = 2 the model reaches the best HOS value

Experiments over different source-target pairs				
Source	Target	OS*	UNK	HOS
Art	Clipart	0.5117	0.7722	0.6155
	Product	0.5517	0.8242	0.6609
	RealWorld	0.6045	0.8016	0.6892
Clipart	Art	0.3262	0.8623	0.4733
	Product	0.4319	0.8286	0.5678
	RealWorld	0.4286	0.849	0.5697
Product	Clipart	0.3667	0.7016	0.4817
	Art	0.4673	0.7427	0.5736
	RealWorld	0.6039	0.7767	0.6795
RealWorld	Clipart	0.455	0.8067	0.5819
	Product	0.5459	0.8202	0.6555
	Art	0.5378	0.7601	0.6299

Table 4. Trend of the experiment as the source and target set vary.

4.1.4 Other evaluations

We now present how the proposed model would perform over different combinations of source and target set. As we said, the Office-Home dataset consists in 4 different sub-datasets: Art, Clipart, Products and RealWorld. The chosen hyperparameters correspond to the optimal ones found so far: α_1 = 1, t = 0.5, α_2 = 2, epochs for step 1 = 50 and epochs for step 2 = 30. It is possible that for some source-target pairs, the given threshold may not be suitable because the number of unknown samples may be too small. For this reason, for the pairs **Art-RealWorld**, **Product-Art**, **Product-RealWorld** and **RealWorld-Art** we have looked for another threshold.
Sometimes the HOS metric is higher and sometimes lower but it remains consistent with the model we found. Table 4 collects OS, UNK and HOS metrics sorted by source-target pair.

4.2. Flip and jigsaw self-supervised tasks

We now move on, looking for other self-supervised tasks that can be implemented in order to solve this problem. The proposed solution works with the Rotation self-supervised

Final metrics by α_2 for Flip task			
OS*	UNK	HOS	α_2
0.494	0.7941	0.6091	0.1
0.4965	0.7971	0.6119	1
0.5035	0.79	**0.615**	**2**
0.4848	0.7788	0.5976	4
0.4748	0.79	0.5932	8
0.222	0.8675	0.3536	16

Table 5. Target samples recognized as known by different threshold values using the Flip self-supervised task.

Final metrics by α_2 for Jigsaw task			
OS*	UNK	HOS	α_2
0.2492	0.7796	0.3777	0.1
0.4679	0.8844	**0.612**	**1**
0.2845	0.8135	0.4216	2
0.3472	0.8835	0.4985	4
0.2053	0.808	0.3274	8
0.2899	0.837	0.4306	16

Table 6. Target samples recognized as known by different threshold values using the Jigsaw self-supervised task.

task but the architecture can be adapted to different types of challenges. In our experiments, we choose to implement the Flip task and the Jigsaw puzzle task taking inspiration from [6].

4.2.1 Flip self-supervised classifier

We already discussed in chapter 3 about the goal of the flip task: the model must be capable in guessing if the image has been vertically flipped or not.
The following model has been trained using a different hyperparameter set. More precisely, we found that for this task the optimal α_1 is 0.1, *t* is 0.57 and α_2 is 2. The optimal number of epochs found for step 1 is 60 and 30 for the step 2. Table 5 shows the results produced by running the task. As we can notice, the flip task works pretty much like the rotation task.

4.2.2 Jigsaw self-supervised Classifier

By performing the jigsaw self-supervised task, the model should learn what permutation of patches has been applied onto the image.
As already said in chapter 3, the original image is divided in patches using a 3x3 grid and the number of jigsaw permutations is 7 and these are the ones having the higher hamming distance according to [7]. By looking at results, we notice that this task is less effective than the others (for each α_2 different from 1). Maybe, by performing further studies on the optimal number of jigsaw permutations, we could land on a better model. Furthermore, it is particularly difficult to reproduce the same exact results, even if the model trained at step 1 was the same from one run to the next. This can be explained by the fact that at each iteration the permutation of the patches is random, thus sometimes it could be easier to predict and sometimes harder.
Table 6 summarizes the results found for this task. Again, the model has been trained from scratch in order to find the optimal one: α_1 = 8, *t* = 0.8, α_2 = 1, the number of epochs for step 1 is 40 and the number of epochs for step 2 is 30.

5. Multi-head architecture: experiments

In this chapter we analyze performances for each task implemented to support the multi-head architecture: in few words, Bucci et al work [1] shows that with a variety of classes, the rotation self-supervised task works better if we modify, in the first step, the number of output neurons of the self-supervised task classifier. More precisely, the multi-head architecture consists in changing, in case of the rotation task, from 4 to $(|C_s|*4)+3$ neurons, where $|C_s|$ stands for the number of classes belonging to the source set, 4 corresponds to the number of possible random rotation applicable to the original image and 3 the highest rotation index. For instance, if the image belongs to the object class *c* = 7 and a 180 degree rotation has been applied to it (rotation index *i* = 2), the new label representing the rotation index is $z = 7 * 4 + 2 = 30$.
It is important mentioning that in step 2 the self-supervised head is changed again with the classic 4 neurons architecture, as shown again by Bucci et al.[1]
Below we show all the results we found, starting from the rotation task, following the flip task and finally the jigsaw task.

5.0.1 Multi-head rotation self-supervised Classifier

In order to expose the final results collected on the Art-Clipart pair, we first present the new scores calculated using the new architecture over 80 epochs of training. Again, we did an early stopping around 50 epochs for the same reason explained in subsection 4.1.1.
You can notice, in table 7, that now scores are pretty higher than the ones computed using the base model (table 1). Table 8 shows the number of samples considered as known by varying the threshold *t* and table 9 contains the final metrics sorted by α_2.
It is worth noting that despite the 13-15% increase in the AUROC score, the final model does not show such evident HOS variations for the highest value. For values between 1 and 4, the new method shows results very similar from the ones provided by the old base model and for very high val-

AUROC score by α_1		
Epochs	AUROC	α_1
80	0.6400	0.1
	0.6464	1
	0.6545	**2**
	0.6366	4
	0.6352	8
	0.6411	16
50	0.6373	2

Table 7. AUROC scores by multi-head Rotation task

Number of known samples by threshold *t*	
Known samples	*t*
4016	0.1
3440	0.15
3187	**0.17**
1973	0.3
870	0.5
316	0.7
54	0.9

Table 8. Target samples recognized as known by different threshold values.

Final metrics by α_2 for multi-head Rotation task			
OS*	UNK	HOS	α_2
0.5356	0.7721	**0.6425**	**0.1**
0.5504	0.7377	0.6304	1
0.5365	0.7426	0.623	2
0.5053	0.7439	0.6018	4
0.0	0.9988	0.0	8
0.0	0.9988	0.0	16

Table 9. Final results for multi-head Rotation task, sorted by HOS metric.

ues, such as 8 or 16, the HOS metric drops to 0 and UNK tends to 1. It may be due to the fact already explained in section 4.1.2: by choosing the theoretical optimal threshold like we did, the new datasets used in step 2 could be rich in incorrect labels.

5.0.2 Other multi-head tasks results

For the sake of completeness, we now finally show the final HOS metrics, respectively for the flip and jigsaw classifiers. Again, every result was computed over the Art-Clipart pair. It is interesting notice that for each task the smaller α_2 is, the higher the HOS metric value will be. Table 10 and 11 show the full metrics study.

Final metrics by α_2 for multi-head Flip task			
OS*	UNK	HOS	α_2
0.5429	0.7389	**0.6259**	**0.1**
0.5493	0.7007	0.6158	1
0.4977	0.7426	0.596	2
0.496	0.6958	0.5791	4
0.0	0.9988	0.0	8
0.0	0.9951	0.0	16

Table 10. Final results for multi-head Flip task, sorted by HOS metric.

Final metrics by α_2 for multi-head Jigsaw task			
OS*	UNK	HOS	α_2
0.4954	0.8494	**0.6258**	**0.1**
0.4522	0.8523	0.5909	1
0.4429	0.8774	0.5887	2
0.4355	0.8623	0.5787	4
0.3809	0.8206	0.5203	8
0.0054	0.9311	0.0108	16

Table 11. Final results for multi-head Jigsaw task, sorted by HOS metric.

6. Conclusion

In conclusion, the domain adaptation problem can be tackled by implementing a supporting self-supervised task, helping the model to enrich its knowledge with domain-invariant features. By doing this, the model should be capable of extending its knowledge over different domains. More precisely, in this particular experiment, we tried to perform an Open-Set domain adaptation problem by splitting the problem in two steps: the first one consists in removing the unknown portion of samples in the target set and then, in step two, the model should be able to align the known part of the target set over the source set.
We analyzed three different self-supervised tasks (rotation, flip and jigsaw puzzle) first on the basic version of the architecture and then on its multi-head version. Speaking of the multi-head version, despite a 13-15% performance increase in step 1, we notice that the final metrics are not much higher than the ones computed with the basic model. Maybe this could be related to our threshold choice: since we could have a not perfect separation between known and unknown samples, the advantages of the multi-head model are not fully exploited. In this case, using the average of the normality scores [1] as a threshold could improve the final results.

References

[1] Bucci, S., Loghmani, M. R., Tommasi, T.: On the effectiveness of image rotation for open set domain adap-

tation. In: ECCV (2020)

[2] https://www.hemanthdv.org/officeHomeDataset.html

[3] Komodakis, N., Spyros G.: Unsupervised representation learning by predicting image rotations. In: ICLR (2018)

[4] Xu, J., Xiao, L., Lpez, A.M.: Self-supervised domain adaptation for computer vision tasks. IEEE Access 7, 156694–156706 (2019)

[5] Izhak Golan and Ran El-Yaniv. Deep anomaly detection us- ing geometric transformations. In NeurIPS, 2018.

[6] Carlucci, F.M., D'Innocente, A., Bucci, S., Caputo, B., Tommasi, T.: Domain generalization by solving jigsaw puzzles. In: CVPR (2019)

[7] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In Euro- pean Conference on Computer Vision (ECCV), 2016.

[8] https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py