POLYTECHNIC OF TURIN

MACHINE LEARNING & DEEP LEARNING
HOMEWORK 3

MAURIZIO VASSALLO, S276961

Table of Contents

1 INTRODUCTION 2
2 DATA EXPLORATION 2
3 PREPROCESSING 3
4 MODELS TRAINING 5
4.1 TRAINING WITHOUT ADAPTATION 6
4.1.1 (EXTRA) HYPERPARAMETER OPTIMIZATION 7
4.2 TRAINING WITH ADAPTATION 9
4.2.1 (EXTRA) HYPERPARAMETER OPTIMIZATION 10
5 CONCLUSIONS 12

1 INTRODUCTION

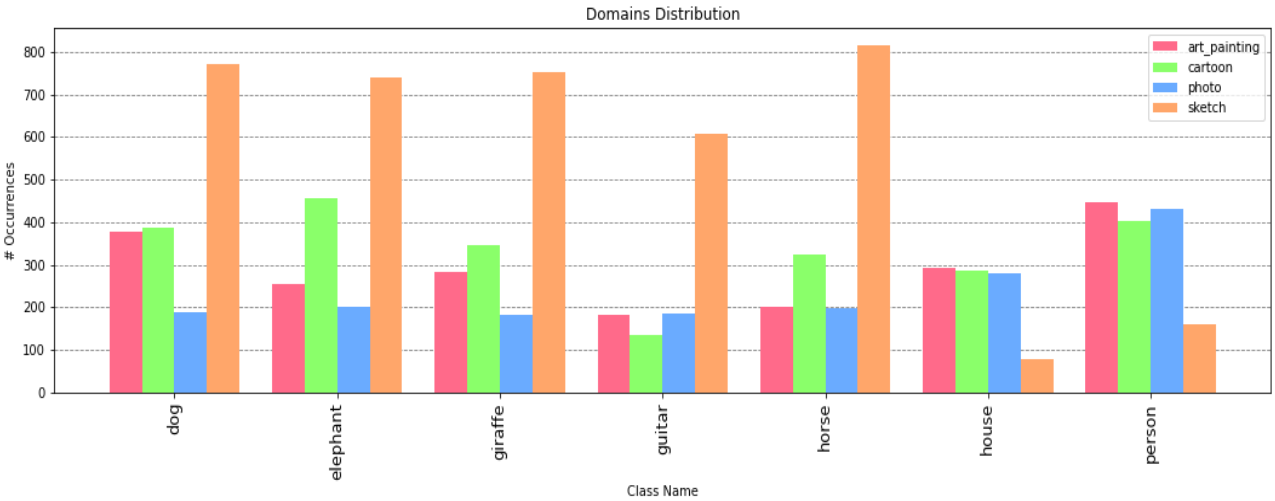
This is the **homework 3** report of the Machine Learning & Deep Learning course at polytechnic of Turin.

The objective is to implement **DANN**, a **Domain Adaptation algorithm**, on the **PACS** dataset using AlexNet.

2 DATA EXPLORATION

This dataset contains 9991 images from 4 different classes, the four domains are: `'art_painting', 'cartoon', 'photo', 'sketch'`. Each domain has 7 categories: `'dog', 'elephant', 'giraffe', 'guitar', 'horse', 'house', 'person'`. The size of each image is roughly 227x227 pixels with 3 channels: RGB.

The distribution of the classes:



Some statistics are:

Stats for **'art_painting'** domain Total Images: 2048 Max # Images in a class: 7 Min # Images in a class: 448 Mean Images for classes: 183.00 STD Images for classes: 291.57	Stats for **'cartoon'** domain Total Images: 2344 Max # Images in a class: 7 Min # Images in a class: 456 Mean Images for classes: 134.00 STD Images for classes: 333.86
Stats for **'photo'** domain Total Images: 1670 Max # Images in a class: 7 Min # Images in a class: 431 Mean Images for classes: 181.00 STD Images for classes: 237.57	Stats for **'sketch'** domain Total Images: 3929 Max # Images in a class: 7 Min # Images in a class: 815 Mean Images for classes: 79.00 STD Images for classes: 560.29

As it is possible to see the categories are quite balanced, even if the sketch domain is more unbalanced with respect to the other classes.

3 PREPROCESSING

For the pre-processing part there are some steps.

Load the data in the different folders. This is done with the *ImageFolder()* method from the *torchvision.datasets* library. At the end the images are divided in the four domains. As requested, the model will train on the photo domain and the target class will be art painting domain.

Also, the images have to be processed using a composition of transformations, in particular data has to be: resized to 256 px, perform a centre crop with 224 px since Alexnet needs an image with 224x224 px size, transform the image in a tensor and normalize this tensor using a specific mean and standard deviation. To know mean and standard deviation means to have a prior knowledge about the dataset or calculate them. In this case will be mean and standard deviation of the ImageNet dataset which are:

```
Mean: (0.485, 0.456, 0.406),
STD: (0.229, 0.224, 0.225).
```

Given the datasets, generate the dataloaders using the *DataLoader()* method from the *torch.utils.data* library. This method divides the dataset in batches given a batch size, in this case 256.
Choosing a good batch size is difficult: the gradient of the loss function is computed over an entire batch. So, a very small batch size means: small amount of data to be storage, fast processing but the gradient would be all over the place and almost random, because learning will be image by image or a small set of them. A larger batch size means more memory to be storage, slower processing but more 'accurate' gradients because now the optimization is over a larger set of images. So, the trade-off is: many 'bad' updates versus few 'good' updates.
I tried different values for the batch size, but Google Colab GPUs limited memory size does not allow too large dimension since it would be out of memory after some epochs. So, I decided to use a batch size of 256, that is not too small actually.

The model needs to be initialized. *Torchvision* library allows to import different models, in this case we are interested in the **Alexnet** model. AlexNet is a convolutional neural network that is 8 layers deep and this has 100 neurons at the last fully connected layer but we have only 7 classes, so we have to change this layer with a 'custom' one using the *Linear(in,out)* method in the *torch.nn* library, the parameters are the input number of neurons (from the previous layer) and the output number of neurons (our number of classes in this case).

For model it is also required another 'path' for domain classifier, this is achieved changing the Alexnet *init()* method adding another *nn.Sequential()* using the same parameter of the already existing classifier, but changing the last layer since we are interested in a binary classification we need only 2 output neurons.

Since the model tries to trick the domain classifier the model has have another parameter *alpha* which is used to train in the right way the domain classifier (positive gradient for the back-propagation) and to train in the 'wrong' way the feature extractor (negative gradient for the back-propagation) in this way the feature extractor gets better a tricking the domain classifier. This parameter alpha must also be optimized so it is added as *nn.Parameter()*.

Also, the classifier and the domain classifier have to have the same initial weights. This is made with changing the weight of one of the two using:

```
net.domain_classifier[i].weight.data = net.classifier[i].weight.data
net.domain_classifier[i].bias.data = net.classifier[i].bias.data
```

The model in this case has:

```
111,595,343 total parameters.
```

Initialized the model, some parameters must be defined:

- Initialize the model with the pre-trained weights,
- A loss function, in this case CrossEntropyLoss function,
- An optimizer, for example Stochastic Gradient Descent which needs to know the parameter to be optimized, the learning rate used, the momentum and the weight decay;
- A scheduler to decrease on a factor *gamma* the learning rate every performed number of steps.

4 MODELS TRAINING

Since the training for deep neural networks is expensive, the models will train on Google GPUs. The GPUs available in Colab often include Nvidia K80s (memory: 24 GB), T4s(memory: 16 GB), P4s (memory: 8 GB) and P100s (memory: 16GB). So, the model before training must be instructed to use GPU, this is made with *model.to('GPU').*

Then the model will train for each epoch on all the batches of the train set, the model will predict the classes of the images and accuracy and loss are calculated comparing the outputs with the labels. Then the loss is back propagated in order to update the weights and improve the performances in the next iteration.

The training model can be divided in 3 steps before calling *optimizer.step()*:

- Forward the training images to the classifier to predict the right category, calculate the loss and back-propagate it,
- Forward the training data to the domain classifier, predict the label (all 0, using *torch.tensor([0] * BATCH_SIZE)*), calculate the loss and back-propagate it,
- Forward the target data to the domain classifier, predict the label (all 1), calculate the loss and back-propagate it.

During this step also loss and accuracies are calculated for 'path': Classifier, Domain Classifier with the train images and Domain Classifier with the target images.

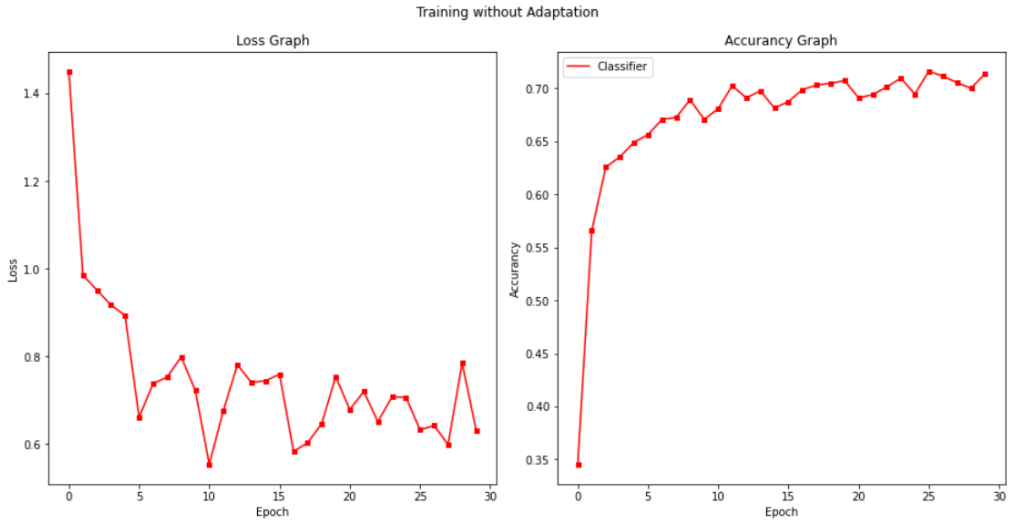
The focus is on learning features that combine, discriminativeness and domain-invariance. This is achieved by jointly optimizing the features extractor as well as two discriminative classifiers operating on: the label classifier that predicts class labels and is used both during training and at test time and the domain classifier that discriminates between the train and the target domains images during training. While the parameters of the classifiers are optimized in order to minimize their error on the training set, the parameters of the feature mapping are optimized in order to minimize the loss of the classifier and to maximize the loss of the domain classifier.

4.1 TRAINING WITHOUT ADAPTATION

The first point is to evaluate the performance of the model on the dataset. The parameter used are:

```
NUM_EPOCHS = 30
LR = 1e-3
MOMENTUM = 0.9
BATCH_SIZE = 256
GAMMA = 0.1
STEP_SIZE = 20
DECAYING_POLICY: decrease the LR of a factor GAMMA after STEP_SIZE
OPTIMIZER = SGD
```

The first time the model is trained without adaptation, it means that the model is a standard AlexNet.




```
Classifier: Min Loss: 0.553737, Max Acc: 0.716168
```

Here are the values for loss and accuracy for the training and evaluation sets.

As expected, the performances are not bad since the dataset the model is pretrained on ImageNet (dataset which contains more than 15 million high-resolution images from 22,000 different categories). The model after 30 epochs seems not to converge yet, so the number of epochs may be increased but since the Google Colab does not give illimited resourced, I did not increase them to avoid use too much power.

The accuracy in the on the target set is:

```
Test Accuracy: 0.4526
```

The score is low since the test set is very different from the training set.

4.1.1 (EXTRA) HYPERPARAMETER OPTIMIZATION

Then, the request is to train from scratch and evaluate the model using different parameters. The chosen parameters are:

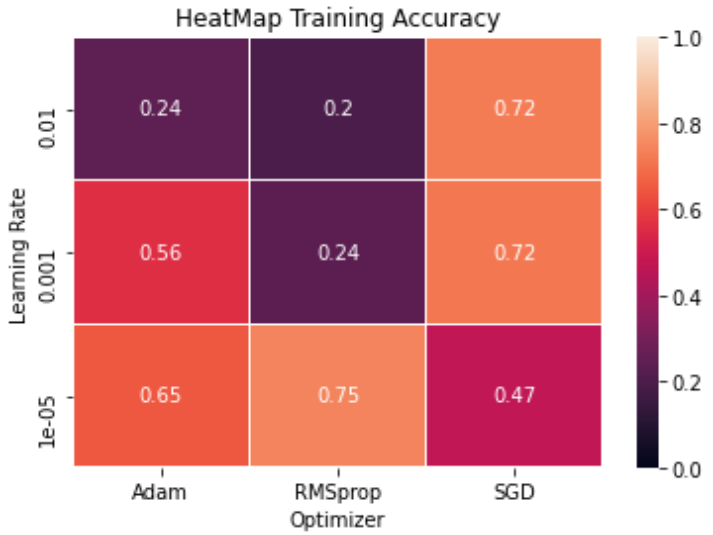
```
LR_values = [0.01, 0.001, 0.00001]
Optimazers = ['RMSprop', 'Adam', 'SGD']
```

The number of epochs is reduced in order to speed up the parameters search. Since usually doing grid search on very deep neural networks is not feasible due to long training time, I decided to train on only 25 epochs also because the model seems to reach its max peak after 15/20 epochs.

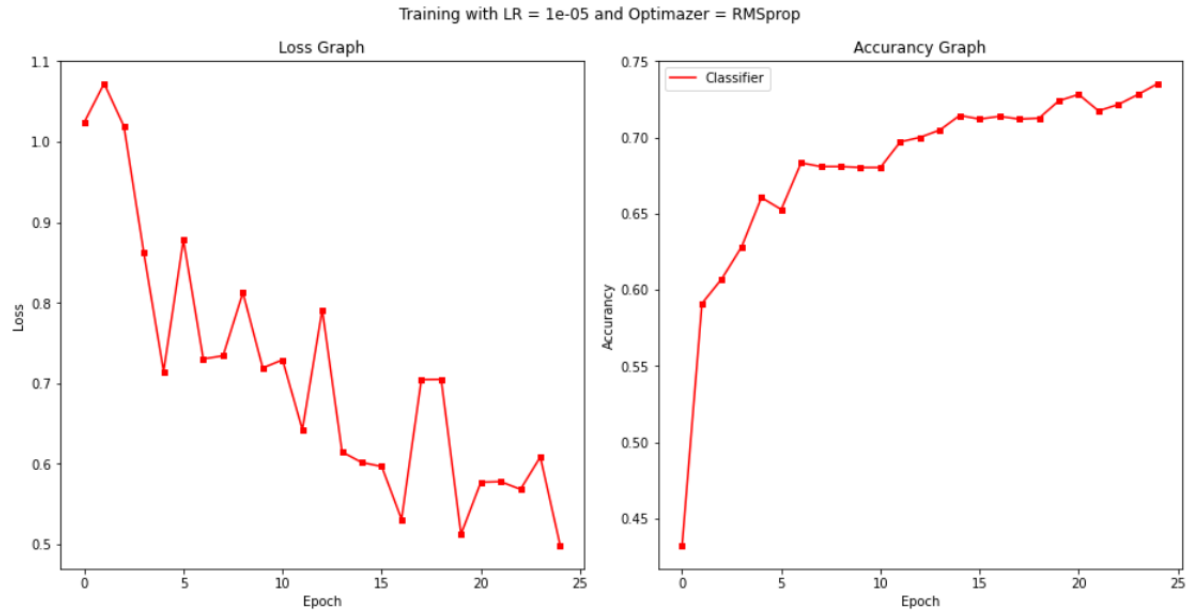
Brief explanation of RMSprop and Adam:

RMSprop is optimization algorithm designed for neural networks. It tries to resolve the problem that gradients may vary in magnitudes: it can be tiny or be huge, which result in difficult updates. It attempts to solve this problem with trying to find a single global learning rate for the algorithm. The algorithm after a full-batch is interested only in the sign of the gradient, with that it can guarantee that all the weights updates are of the same size. With this adjustment it better deals with saddle points and plateaus as since it takes big enough steps even with tiny gradients (This cannot be solved just increasing the learning rate because the steps it takes with large gradients are going to be even bigger, which will result in divergence). So, RMSprop combines the idea of only using the sign of the gradient with the idea of adapting the step size for each weight. The step is adjusted considering the 2 previous gradients for the weights, is they are the same sign it means that that weight is going in the right direction and it should accelerate (increase learning rate), if they are different it must decelerate (decrease learning rate).

Adam is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks. It can be seen as a combination of RMSprop and Stochastic Gradient Descent with momentum, in fact, it uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.
Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters using estimations of first (mean) and second moments (uncentered variance, don't subtract the mean during the calculation of the variance) of gradient to adapt the learning rate for each weight of the neural network.
The main reasons to choose it are: the update of the weights is invariant of the gradient magnitude, which helps a lot in overcome areas with small gradients (like saddle points). In these areas SGD struggles to overcome them and it requires less memory to compute.



Here are the heatmaps of the result for train and validation accuracies for the different pairs LR-Optimizer. The pair who performs better in the validation set is (0.00001, RMSprop).




```
Classifier: Min Loss: 0.498298, Max Acc: 0.735329
```

Here the training graph of best pairs of parameters found.

The accuracy in the on the test set is:

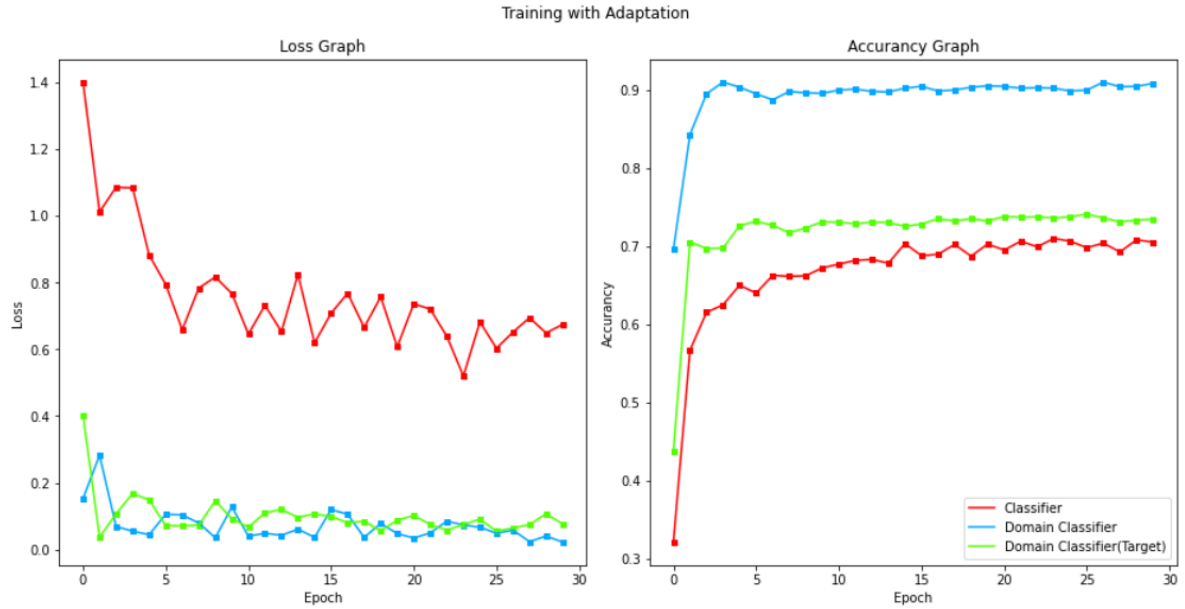
```
Test Accuracy: 0.4155
```

Even if the accuracy is higher that the training with the standard parameters (0.001, SDG), the accuracy on the target set is lower. This may be explained since the other pair of parameters may overfit the training data and so it is not able to generalize well.

4.2 TRAINING WITH ADAPTATION

Now the model is trained using domain adaptation, using the same parameters as before and ALPHA with an initial value of 0.1.

Now the model is a bit different from standard Alexnet structure, since now there is a new ‘branch’, with which we try to improve performances feeding th new branch with both train and target data domain images.




```
Classifier: Min Loss: 0.519971, Max Acc: 0.710180
Domain Classifier: Min Loss: 0.023147, Max Acc: 0.910180
Domain Classifier(Target): Min Loss: 0.037678, Max Acc: 0.741211
```

This is what happens after the full train the model using both training and target samples.

It is possible to see that the model performs better with respect to the model trained without adaptation, in particular it gets a lower loss and a higher accuracy even if the difference is not large.

The accuracy in the on the test set is:

```
Test Accuracy: 0.4653
```

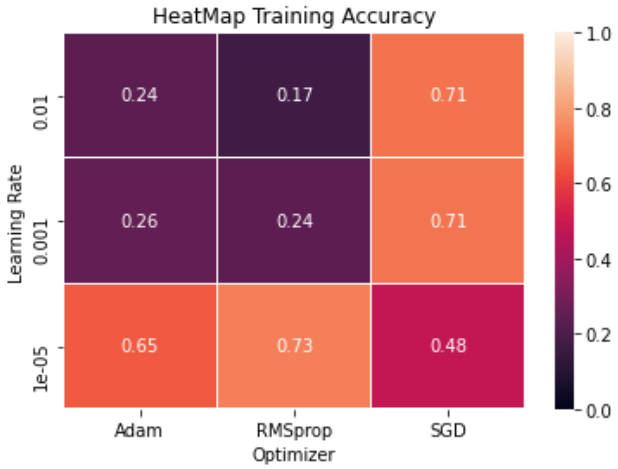
It is possible to see that the accuracy is slightly higher than training without adaptation, even if the difference is small.

4.2.1 (EXTRA) HYPERPARAMETER OPTIMIZATION

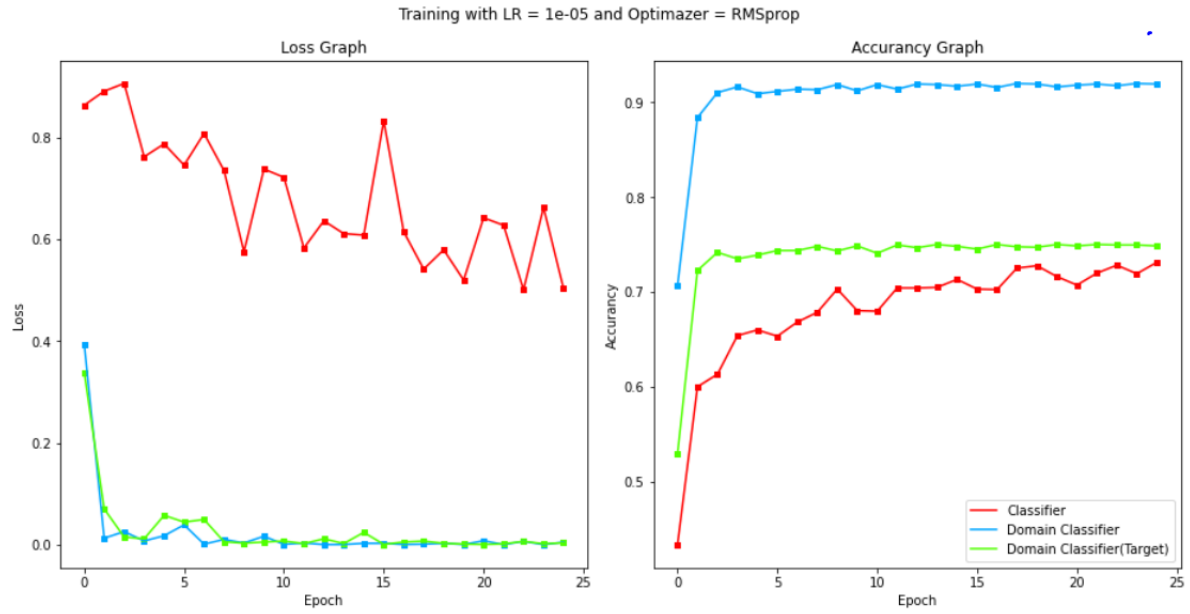
The parameters as the same as before:

```
LR_values = [0.01, 0.001, 0.00001]
Optimizers = ['RMSprop', 'Adam', 'SGD']
```

Evaluating the model with the different pairs, the results are:



It is possible to see the generally the accuracies are higher in the model with adaptation, this means that the adaptation is working properly and it helps the model to generalize. Again, the pairs with best performances is (0.00001, RMSprop).

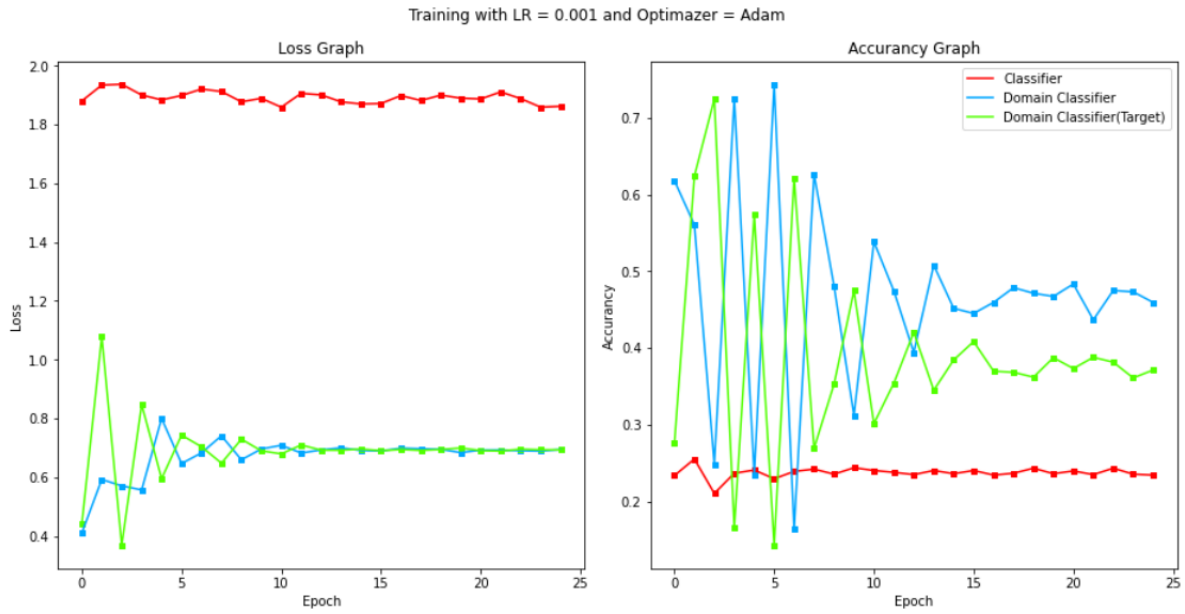



```
Classifier: Min Loss: 0.500660, Max Acc: 0.731138
Domain Classifier: Min Loss: 0.000095, Max Acc: 0.919760
Domain Classifier(Target): Min Loss: 0.000556, Max Acc: 0.750000
```

Here the graph with these parameters, it is possible to see that the losses are generally decreasing, and the accuracies are increasing even if it gets stuck in a local minimum after some epochs.

The accuracy in the on the test set is:

```
Test Accuracy: 0.4521
```

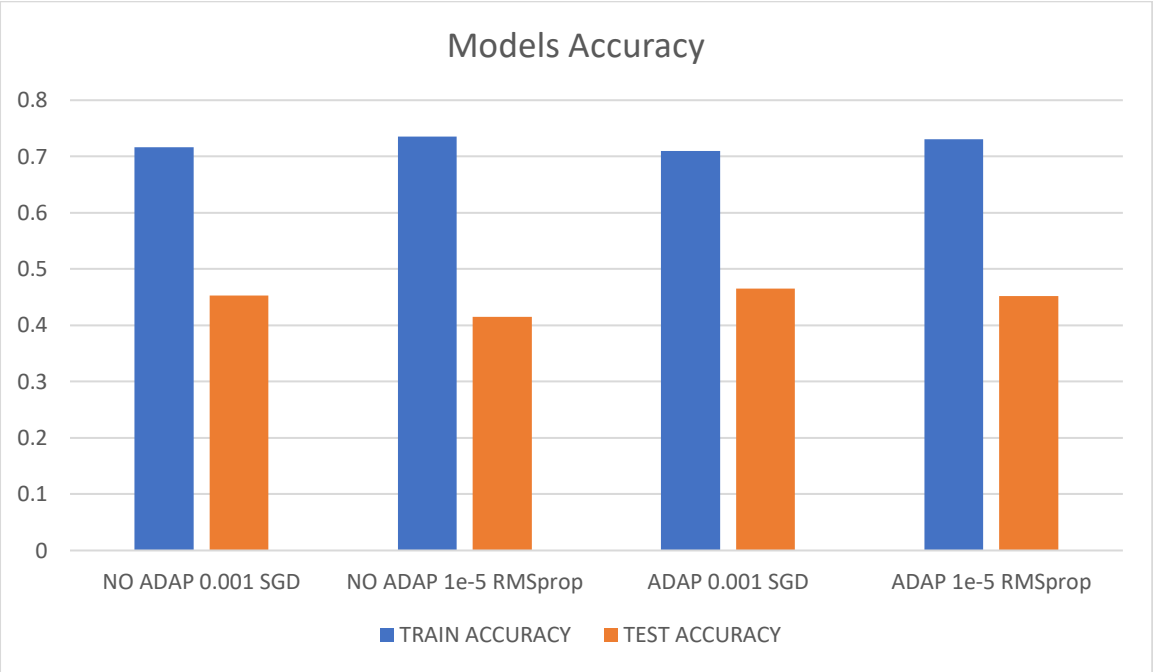



```
Classifier: Min Loss: 1.858608, Max Acc: 0.255090
Domain Classifier: Min Loss: 0.409942, Max Acc: 0.743713
Domain Classifier(Target): Min Loss: 0.368336, Max Acc: 0.724609
```

An interesting graph is the one with pair (0.001, Adam). Here, it is possible to see the conflict between the Domain Classifier and the Domain Classifier (Target), when one increases the other decrease. This can be explained since the model tries to fool the domain classifier using two different datasets, photo and art painting.

5 CONCLUSIONS

From the obtained values it is possible to see that the network with adaptation performs better on the target set. The difference is not very high, 1%-3% point but



MODEL	TRAIN ACCURACY	TEST ACCURACY
NO ADAP 0.001 SGD	0.7161	0.4526
NO ADAP 1e-5 RMSprop	0.7353	0.4155
ADAP 0.001 SGD	0.7101	0.4653
ADAP 1e-5 RMSprop	0.7311	0.4521

This DANN model allows to adapt the knowledge of the net obtained after the training of a specific set on a different set which has some analogies. For example, the model must classify is a review on books is positive or not. The model is trained on the book set but after some time, the model has to evaluate some films reviews that are not labelled as positive or negative (it can be considered as a test set), so it is impossible to train a new model on this set, since it can not access the labels. With DANN, it is possible to use the model trained with the book dataset for evaluating the films set having similar performances.