Artificial Intelligence & Machine Learning
Homework 3

Matteo Stoisa - s265542

Introduction

The aim of the homework is to implement, analyze and optimize the functioning of DANN: a neural network algorithm designed to better classify images of different domain than the training-set one.

In the second homework we analyzed the functioning of the AlexNet classifier in optimal conditions: training images and test images belonged to the same domain so that the classification accuracy was appreciably high. But in many real cases the training-dataset belongs to a domain and the test-dataset belongs to a different domain, for example due to different conditions in which they can be acquired.

I implemented code in Python 3 mainly using PyTorch library and ran it in Colab environment using GPUs which guarantees faster computation.

Dataset

I used PACS dataset, an image classification dataset composed by 7 classes (dog, elephant, giraffe, guitar, horse, house, person) divided into 4 different domains (art painting, cartoon, photo, sketch).

I implemented the DANN starting from AlexNet neural network code with pre-training on ImageNet photo dataset, then I applied a fine-grained training using PACS photos (more than 1600). The tests are performed with PACS art painting images (more than 2000).

Structure

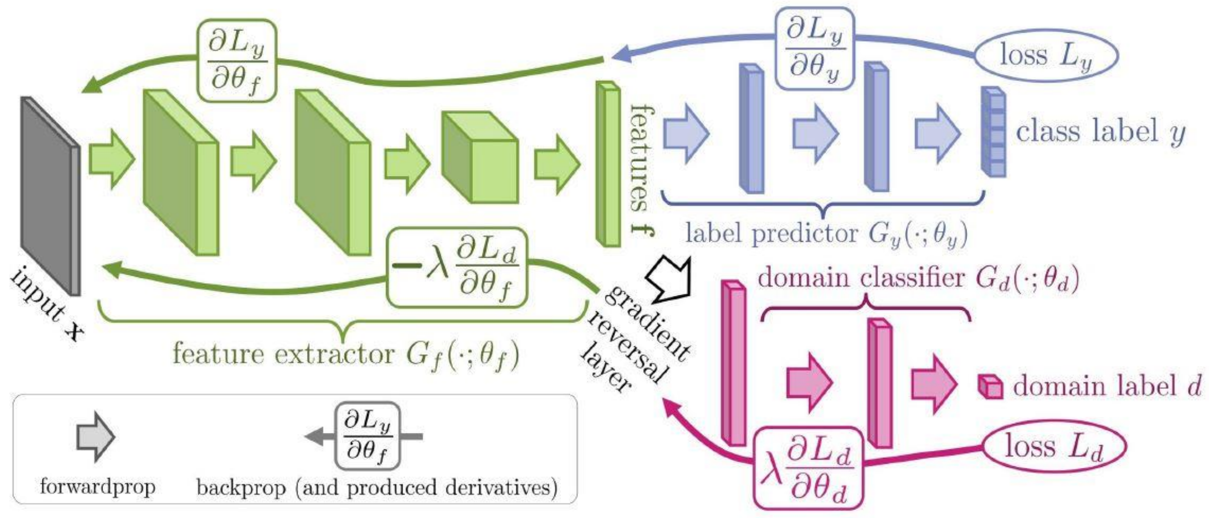
The structure of AlexNet is composed by five convolutional layers and three fully-connected layers that predict the class of a certain image. The idea of the DANN is to insert in the neural network another classifier that predict the domain of the image.

More precisely, during the training procedure a train-image passes through convolutional layers so that features are extracted with the standard forward-propagation mechanism, then features are passed to the standard classifier that predicts the class trying to minimize his loss and updating weights.

In the DANN, features are also passed to the discriminator: a binary classifier (similar in structure to the standard AlexNet fully-connected layer classifier) that tries to tell if the image comes from the train-domain or from the test-domain. During the train procedure images from both domains are passed to the discriminator as supervised learning aimed to fool it. In the final ideal case, it is impossible for the discriminator to distinguish the domain of the image, this would mean that the classifier is now able to classify in the same way, and with higher accuracy, images from both domains.

To achieve this result, it is applied to the network a back-propagation update aimed to decrease the classifier loss and, at the same time, to increase the discriminator loss. The latter is calculated as the inversion of the standard learning gradient and multiplied by a coefficient `ALPHA` (between 0 and 1).

In the test procedure the discriminator is no longer used.


$\frac{\partial L_y}{\partial \theta_f}$
$\frac{\partial L_y}{\partial \theta_y}$
loss L_y
class label y
features **f**
label predictor $G_y(\cdot;\theta_y)$
$-\lambda\frac{\partial L_d}{\partial \theta_f}$
input x
feature extractor $G_f(\cdot;\theta_f)$
gradient reversal layer
domain classifier $G_d(\cdot;\theta_d)$
domain label d
loss L_d
$\lambda\frac{\partial L_d}{\partial \theta_d}$
forwardprop
backprop (and produced derivatives)

Implementation

Starting from AlexNet implementation, here are the main steps I have applied to implement the DANN:

I added the domain discriminator (similar to the class classifier, with a last-layer of two dimension):

```
#DOMAIN DISCRIMINATOR
self.discriminator = nn.Sequential(
  nn.Dropout(),
  nn.Linear(256 * 6 * 6, 4096),
  nn.ReLU(inplace=True),
  nn.Dropout(),
  nn.Linear(4096, 4096),
  nn.ReLU(inplace=True),
  nn.Linear(4096, 2),
)
```

I modified the forward function in order to call the gradient revert in case of discriminator back-propagation:

```
def forward(self, x, alpha=None):
  x = self.features(x)
  x = self.avgpool(x)
  x = torch.flatten(x, 1)
  x = x.view(x.size(0), -1) #?
  if alpha is not None: #forward in domain discriminator
    reverse_feature = ReverseLayerF.apply(x, alpha)
    x = self.discriminator(reverse_feature)
  else: #forward in class classifier
    x = self.classifier(x)
  return x
```

```
class ReverseLayerF(Function):
  @staticmethod
  def forward(ctx, x, alpha):
    ctx.alpha = alpha
    return x.view_as(x)
  @staticmethod
  def backward(ctx, grad_output):
    output = grad_output.neg() * ctx.alpha
    return output, None
```

After importing AlexNet pre-trained on ImagNet, I copied the pre-trained weights of the classifier's fully-connected layers into the blank discriminator's ones. The last classifier layer dimension is set to 7 (number of PACS classes):

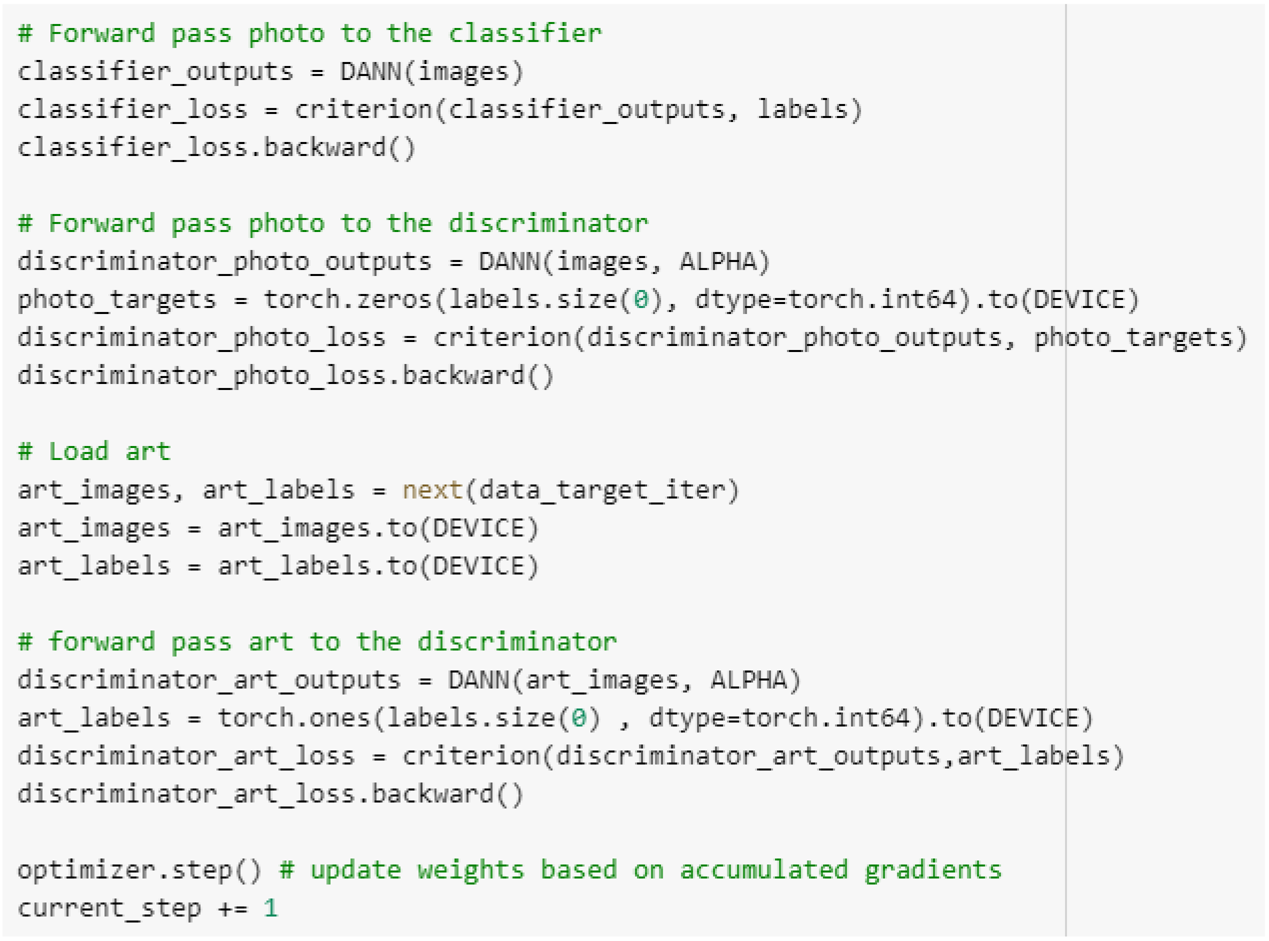
```
def prepareDANN():
  net = alexnet(pretrained=True)

  net.discriminator[1].weight.data = net.classifier[1].weight.data
  net.discriminator[1].bias.data = net.classifier[1].bias.data

  net.discriminator[4].weight.data = net.classifier[4].weight.data
  net.discriminator[4].bias.data = net.classifier[4].bias.data

  net.classifier[6] = nn.Linear(4096,7)
  return net
```

Here are the main steps done each epoch during the train phase: forward the train-domain-image to the classifier and to the discriminator, load a test-domain-image and forward it to the discriminator, then update the network with all calculated gradients:



```
# Forward pass photo to the classifier
classifier_outputs = DANN(images)
classifier_loss = criterion(classifier_outputs, labels)
classifier_loss.backward()

# Forward pass photo to the discriminator
discriminator_photo_outputs = DANN(images, ALPHA)
photo_targets = torch.zeros(labels.size(0), dtype=torch.int64).to(DEVICE)
discriminator_photo_loss = criterion(discriminator_photo_outputs, photo_targets)
discriminator_photo_loss.backward()

# Load art
art_images, art_labels = next(data_target_iter)
art_images = art_images.to(DEVICE)
art_labels = art_labels.to(DEVICE)

# forward pass art to the discriminator
discriminator_art_outputs = DANN(art_images, ALPHA)
art_labels = torch.ones(labels.size(0) , dtype=torch.int64).to(DEVICE)
discriminator_art_loss = criterion(discriminator_art_outputs,art_labels)
discriminator_art_loss.backward()

optimizer.step() # update weights based on accumulated gradients
current_step += 1
```

Tests

I evaluated the DANN operation with different hyperparameter in order to analyze the behavior and to find the best settings.

For each set of parameters, I have at first tested the standard AlexNet without domain adaptation: starting from the pre-trained network on ImageNet I have performed a fine-grained training with photos and then performed the test on art painting. Using the obtained classification accuracy as comparison, I have performed the training of the DANN (always starting from ImageNet pre-training) with photos and adaptation on art painting, then calculated classification accuracy on art painting.

I decided to keep the following AlexNet hyperparameters constant:

BATCH_SIZE =	128
MOMENTUM =	0.9
WEIGHT_DECAY =	5e-5
GAMMA =	0.1

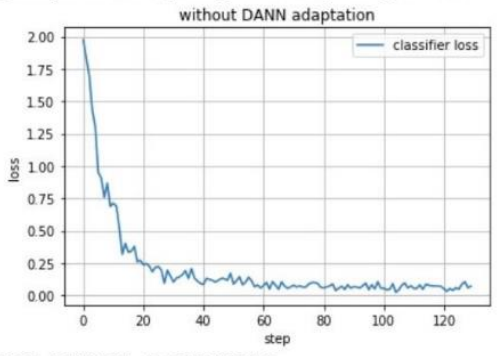
For the following hyperparameters, I initially performed some test with manual tries and then, starting from observations done on them, I have implemented some focused grid-search:

`LR`	`(starting learning rate)`
`NUM_EPOCH`	`(number of training epochs)`
`STEP_SIZE`	`(number of epochs after that the LR is decreased by factor 10)`
`ALPHA`	`(coefficient of the discriminator back-propagation)`

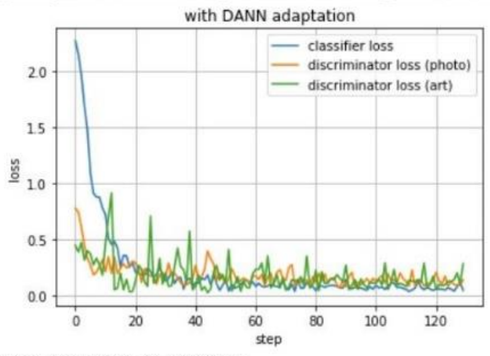
Analysis

Following graphs show the trend of classifier loss and discriminator loss (on photos and on art paintings) over training epochs. For each training procedure is then calculated the accuracy score on the test without (first) and with domain adaptation.

```
[0/36] LR: 0.0005, NUM_EPOCHS: 10, STEP_SIZE: 6
```



```
TEST ACCURACY: 0.50537109375
[1/36] LR: 0.0005, NUM_EPOCHS: 10, STEP_SIZE: 6, ALPHA: linearly ascendant
```

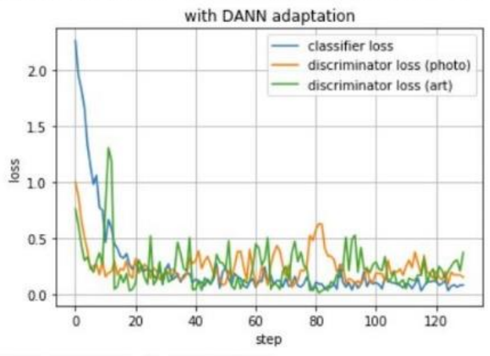


```
TEST ACCURACY: 0.48828125
```

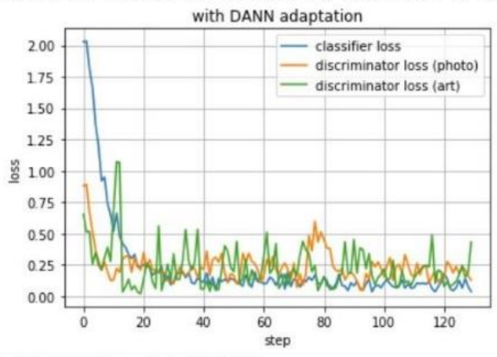
As first observation `LR=0.0005` is the bigger value of learning rate that doesn’t cause the network to diverge. Higher values I have tried make the classifier loss grow enormously and the accuracy score stay under 20%.

The standard AlexNet scores a classification accuracy of 50%, the three DANN varying `ALPHA={linear, 0.1, 0.3}` score 1 or 2 percentage point less.

```
[2/36] LR: 0.0005, NUM_EPOCHS: 10, STEP_SIZE: 6, ALPHA: flat 0.1
```



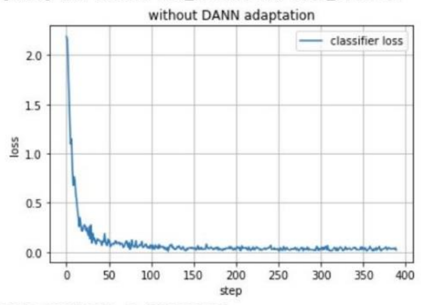
```
TEST ACCURACY: 0.48681640625
[3/36] LR: 0.0005, NUM_EPOCHS: 10, STEP_SIZE: 6, ALPHA: flat 0.3
```



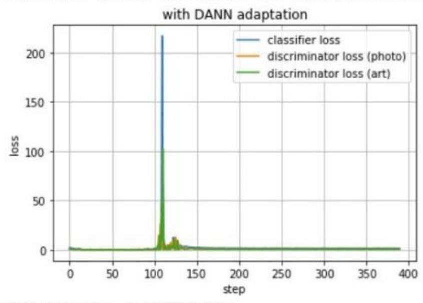
```
TEST ACCURACY: 0.4736328125
```

Furthermore, a huge number of training epochs does not improve the network performance but in many cases cause it to degenerate, the following tests are performed with `LR=0.0005` `ALPHA={linear, 0.1, 0.3}` and varying `NUM_EPOCH` and `STEP_SIZE` to `{(30, 10), (50, 20)}`:

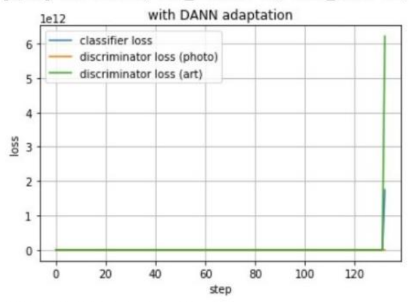
```
[4/36] LR: 0.0005, NUM_EPOCHS: 30, STEP_SIZE: 10
```



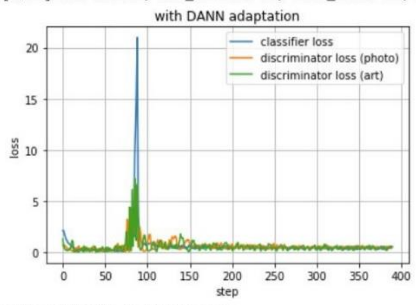
```
TEST ACCURACY: 0.4775390625
[5/36] LR: 0.0005, NUM_EPOCHS: 30, STEP_SIZE: 10, ALPHA: linearly ascendant
```



```
TEST ACCURACY: 0.18115234375
[6/36] LR: 0.0005, NUM_EPOCHS: 30, STEP_SIZE: 10, ALPHA: flat 0.1
```

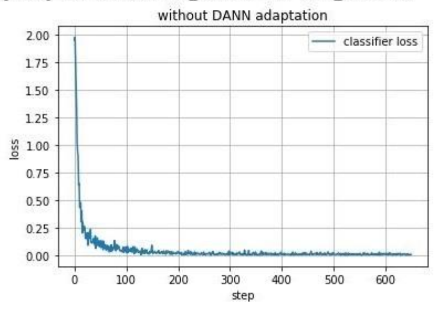


```
TEST ACCURACY: 0.18505859375
[7/36] LR: 0.0005, NUM_EPOCHS: 30, STEP_SIZE: 10, ALPHA: flat 0.3
```

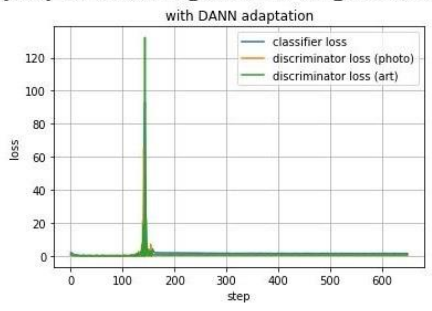


```
TEST ACCURACY: 0.41259765625
```

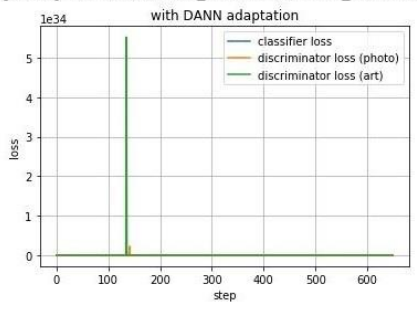
```
[8/36] LR: 0.0005, NUM_EPOCHS: 50, STEP_SIZE: 20
```



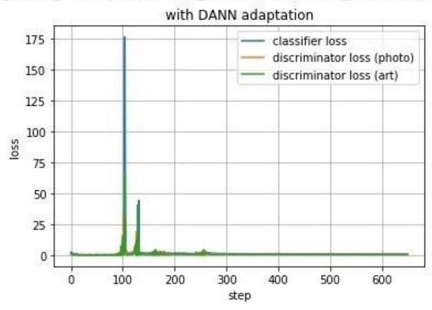
```
TEST ACCURACY: 0.4521484375
[9/36] LR: 0.0005, NUM_EPOCHS: 50, STEP_SIZE: 20, ALPHA: linearly ascendant
```



```
TEST ACCURACY: 0.16552734375
[10/36] LR: 0.0005, NUM_EPOCHS: 50, STEP_SIZE: 20, ALPHA: flat 0.1
```



```
TEST ACCURACY: 0.14404296875
[11/36] LR: 0.0005, NUM_EPOCHS: 50, STEP_SIZE: 20, ALPHA: flat 0.3
```

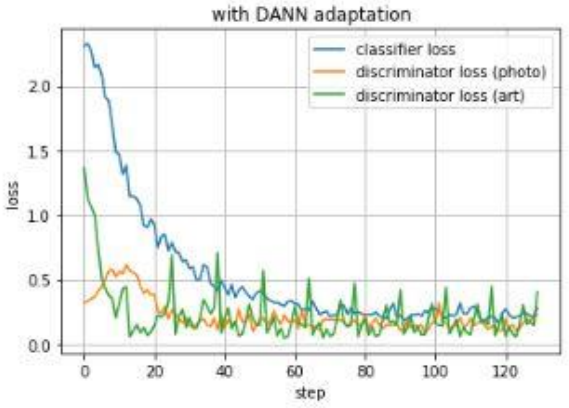


```
TEST ACCURACY: 0.26025390625
```

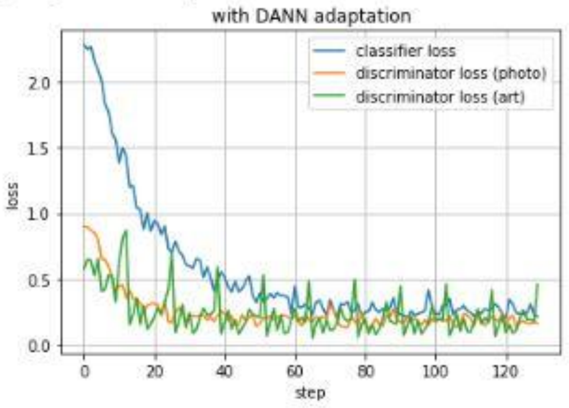
I consequently decided to keep `NUM_EPOCH` and `STEP_SIZE` constant to `(10, 6)`.

Using LR=0.0001 and varying ALPHA={linear, 0.1, 0.5, 1} the standard Alexnet scores only 40% accuracy but the DANN adaptations improve the score from 3 to 5 percentage points.

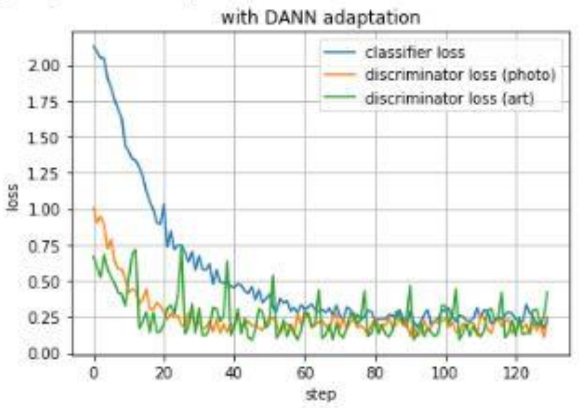
```
[5/20] LR: 0.0001
TEST ACCURACY (without DANN adaptation): 0.40185546875
[6/20] LR: 0.0001, ALPHA: linear
```



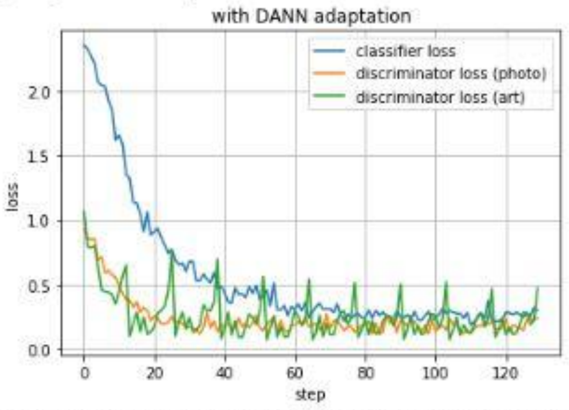
```
TEST ACCURACY (with DANN adaptation): 0.435546875
[7/20] LR: 0.0001, ALPHA: flat 0.1
```



```
TEST ACCURACY (with DANN adaptation): 0.4521484375
[8/20] LR: 0.0001, ALPHA: flat 0.5
```



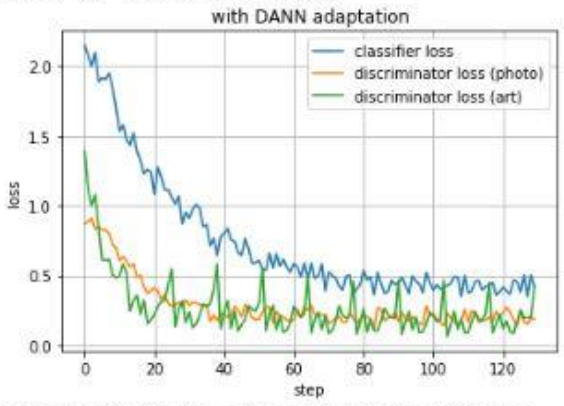
```
TEST ACCURACY (with DANN adaptation): 0.44482421875
[9/20] LR: 0.0001, ALPHA: flat 1
```



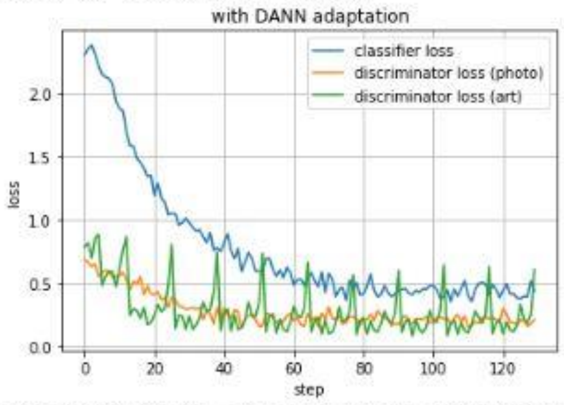
```
TEST ACCURACY (with DANN adaptation): 0.43115234375
```

With ALPHA=0.00005 and varying ALPHA={linear, 0.1, 0.5, 1} the standard Alexnet scores less than 40% accuracy, the best DANN adaptations score up to 9 percentage points better (in the case of linear ALPHA).

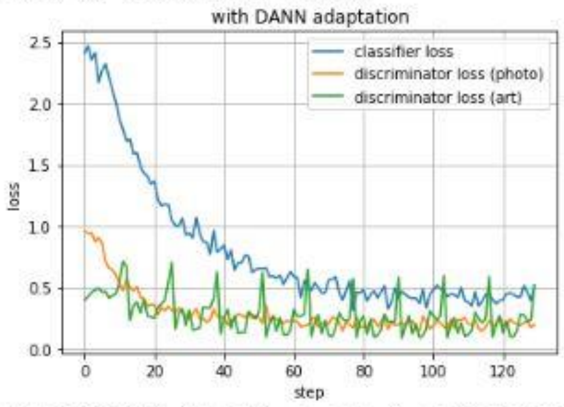
```
[0/20] LR: 5e-05
TEST ACCURACY (without DANN adaptation): 0.37939453125
[1/20] LR: 5e-05, ALPHA: linear
```



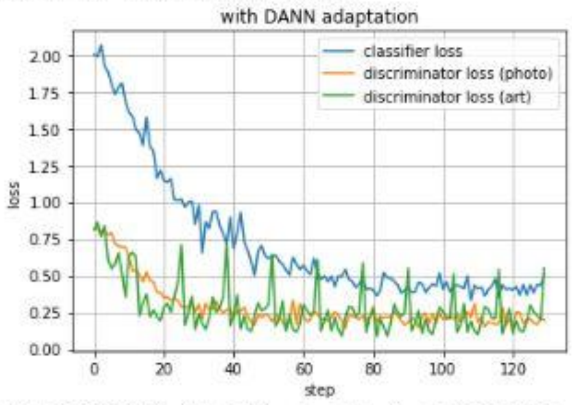
```
TEST ACCURACY (with DANN adaptation): 0.4609375
[2/20] LR: 5e-05, ALPHA: flat 0.1
```



```
TEST ACCURACY (with DANN adaptation): 0.41943359375
[3/20] LR: 5e-05, ALPHA: flat 0.5
```



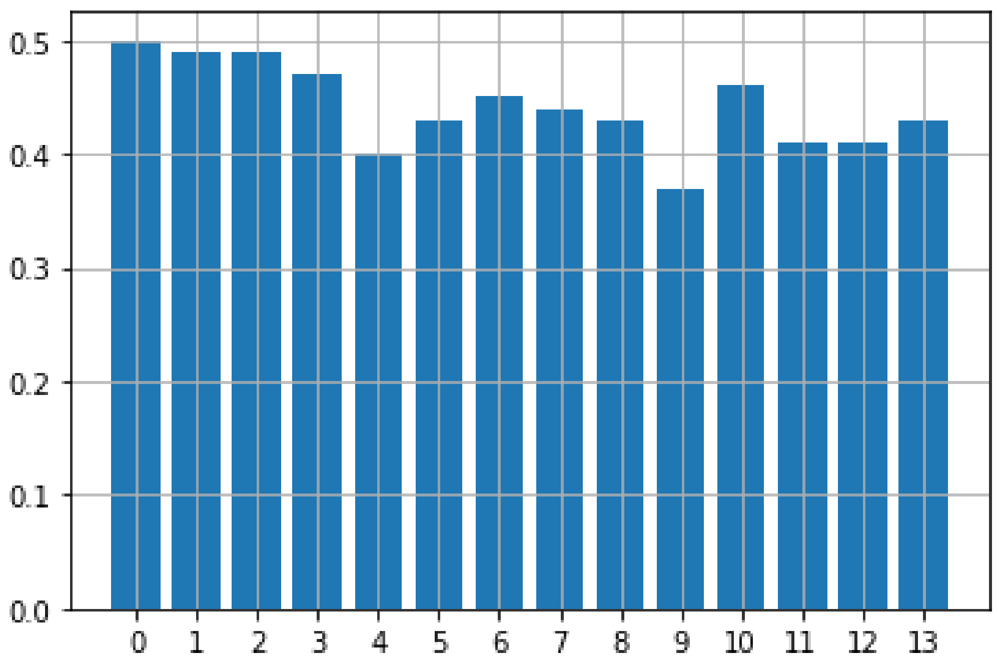
```
TEST ACCURACY (with DANN adaptation): 0.4169921875
[4/20] LR: 5e-05, ALPHA: flat 1
```



```
TEST ACCURACY (with DANN adaptation): 0.439453125
```

Here are reported the classification accuracy score achieved in the most significant tests:

```
0:  LR=0.0005  (standard AlexNet)    0.5
1:  LR=0.0005  ALPHA=linear          0.49
2:  LR=0.0005  ALPHA=0.1             0.49
3:  LR=0.0005  ALPHA=0.3             0.47
4:  LR=0.0001  (standard AlexNet)    0.44
5:  LR=0.0001  ALPHA=linear          0.43
6:  LR=0.0001  ALPHA=0.1             0.45
7:  LR=0.0001  ALPHA=0.5             0.44
8:  LR=0.0001  ALPHA=1               0.43
9:  LR=0.00005 (standard AlexNet)    0.37
10: LR=0.00005 ALPHA=linear          0.46
11: LR=0.00005 ALPHA=0.1             0.41
12: LR=0.00005 ALPHA=0.5             0.41
13: LR=0.00005 ALPHA=1               0.4
```



The higher accuracy score is achieved without DANN adaptation, on this particular study case with this dataset and pre-trained network the domain adaptation is not necessary.

Nevertheless, in many cases the adaptation improves the accuracy of many percentage points, this leaves you thinking that the domain adaptation could be considerably relevant in some case with other peculiarity.