The dataset

After an initial analysis of the dataset, we can start by loading the source and target data using *ImageNet*:

```
# Clone github repository with data
if not os.path.isdir('./Homework3-PACS'):
  !git clone https://github.com/MachineLearning2020/Homework3-PACS.git

DATA_DIR = 'Homework3-PACS/PACS'

# Prepare Pytorch train/test Datasets
train_dataset = torchvision.datasets.ImageFolder(DATA_DIR+'/photo', transform=train_transform)
test_dataset = torchvision.datasets.ImageFolder(DATA_DIR + '/art_painting', transform=
    eval_transform)
```

Those are taken from two different domains: it is needed to load the *photo* folder as source domain for the train and the *art_painting* one as target domain and test.

Implementing the model

Task A

In order to build the *DANN AlexNet* it has been added a new classifier named `dann_classifier` in the *__init__()* function of our class named *RandomNetworkWithReverseGrad*:

```
# OTHER LAYERS

self.dann_classifier = 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, 1000),
)

#OTHER LAYERS
```

That is exactly a copy of the AlexNet classifier. The last `nn.Linear` layer is left with an output of 1000 because of ImageNet (in order to copy the weights afterwards).
The last `nn.Linear` of *dann_classifier* and *classifier* are modified to fit the 7 classes for the second one and 2 (the two domains) for the first one.

```
net.classifier[6] = nn.Linear(4096, 7)
net.dann_classifier[6] = nn.Linear(4096, 2)
```

Task B

Now it is time to copy the weights of the pretrained model from the `classifier` to the new `dann_classifier`: in our *alexnetdann()* function is added a for-loop that takes the data from weights and biases of the G_y and copies it on the G_d:

```python
def alexnetdann(pretrained=False, progress=True, **kwargs):
    model = RandomNetworkWithReverseGrad(**kwargs)
    if pretrained:
        state_dict = load_state_dict_from_url(model_urls['alexnet'], progress=progress)
        model.load_state_dict(state_dict, strict=False)  # Set strict to False to avoid errors

        # Copy the weights and data from the AlexNet classifier to the dann one
        for i in [1, 4]:
          model.dann_classifier[i].weight.data = model.classifier[i].weight.data
          model.dann_classifier[i].bias.data = model.classifier[i].bias.data

    return model
```

In particular the second and are the only nn.Linear layers to be copied.

Task C

At last, we must modify the *forward()* function inherited from `nn.Model` such that, when the parameter `alpha` is passed, the batch must be forwarded to G_d (to G_y otherwise).

```python
def forward(self, x, alpha=None):
    features = self.features(x)
    # Flatten the features:
    features = features.view(features.size(0), -1)
    # If we pass alpha, we can assume we are training the discriminator
    if alpha is not None:
        # gradient reversal layer (backward gradients will be reversed)
        reverse_feature = ReverseLayerF.apply(features, alpha)
        discriminator_output = self.dann_classifier(reverse_feature)
        return discriminator_output
    # If we don't pass alpha, we assume we are training with supervision
    else:
        class_outputs = self.classifier(features)
        return class_outputs
```

In this way, when we are training the discriminator it is applied the gradient reverse layer and then forwarded to our discriminator G_d. Instead, when we are training the classifier it is only forwarded to the `classifier` layers. This architecture tries to fool the features extractor G_f so that it will be trained on features not dependent on the domain itself.

Domain adaptation

Task A

Now that our model has been designed, it is time to apply it. For this first task, we trained the AlexNet model without adaptation on the *photos* domain (pretrained on ImageNet) and tested it on the *art_painting* one.
With a batch size of 256, a learning rate of 0.0001 and 10 epochs we obtain an accuracy of 41%, that demonstrates our domain adaptation problem: we cannot say for sure that our model will perform good in real life tasks that shifts the domain (near all of them).
In order to do it, we simply trained on the *dataloader* that iterates batch-by-batch the *train_dataset* forwarding only the features without the alpha parameter so that we use the original AlexNet net.

Task B

For this second task, we are going to apply the domain adaptation to get better results on our test set. We train simultaneously the classifier on the labeled source, the discriminator on the unlabeled source and unlabeled target. This three steps are done sequentially in one iteration of the *dataloader*:

```
for i, (images, labels) in enumerate(train_dataloader):
    # ** TRAIN ON SOURCE LABELS **

    # ** TRAIN THE DISCRIMINATOR ON SOURCE **

    # ** TRAIN THE DISCRIMINATOR ON TARGET **

    optimizer.step()
```

In particular, the first step is the standard one used several time for training (and in the previous task). The second one and the third are built as follows:

```
# ** TRAIN THE DISCRIMINATOR ON SOURCE **

# Forward pass to the network
outputs = net(images, alpha=alpha)

labels_zeros = torch.zeros(labels.shape[0]).type(torch.LongTensor).to(DEVICE)

# Compute loss based on output and zero-labels
loss_source_dann = criterion_dann(outputs, labels_zeros)

loss_source_dann.backward()

# Log loss
if i % LOG_FREQUENCY == 0:
  print('2 Step {}, Loss {}'.format(current_step, loss_source_dann.item()))
```

```
# ** TRAIN THE DISCRIMINATOR ON TARGET **

try:
  images2, _ = next(dataloader_iterator)
except StopIteration:
  print("EXEPT")
  dataloader_iterator = iter(train_dataloader_targetdomain)
  images2, _ = next(dataloader_iterator)

labels_ones = torch.ones(labels.shape[0]).type(torch.LongTensor).to(DEVICE)

# Forward pass to the network
outputs = net(images2, alpha=alpha)

# Compute loss based on output and one-labels
loss_target = criterion_dann(outputs, labels_ones)

loss_target.backward()

# Log loss
if i % LOG_FREQUENCY == 0:
  print('3 Step {}, Loss {}'.format(current_step, loss_target.item()))
```

We can see that for the source training it is used the same batch used by the first step, while for the target training it is implemented an manual iterator on the test set that discards the labels too. The activation of the training on the discriminator is triggered by the specification of the alpha parameter, as implemented in the first tasks.
For this task, they have been chosen the same hyperparameters, with an adaptive alpha retrieved from the paper of Ganin et al. that - depending on the current epoch, totale epochs, current step and others - goes from 0 to 1: the results is an accuracy of nearly 47%, so a good gain w.r.t. the net without adaptation.
From the Fig.1 we can see the three losses. It has been chosen a low number of epochs for time saving, but leaving the epochs to 50-100 would lead to an average increase of the target discriminator loss.

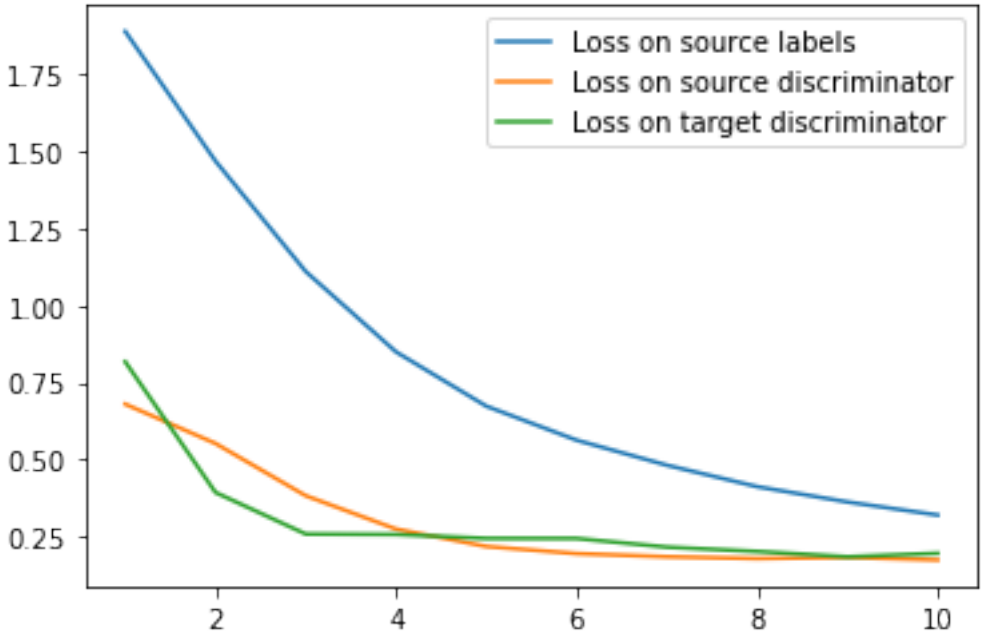

Figure 1: Losses evolution

Task C

Looking at those accuracies, we can say that the DANN, on average, does give better results. All of this is achieved because of the discriminator's reversal layer that lead the features extractor to maximize the error of the discriminator and tries to fool it generating domain invariant features on which the classifier is trained on, giving better results. The dataset is relatively small and there are not so much possibilities of improvements, but with bigger ones (like MNIST with other domains) it is possible to achieve even better.

Cross domain validation

The validation task applied to the DANNs is still a research topic. It can be possible to implement a *cross domain validation* training on `photos` and testing on other domains to do some hyperparameters tuning (in particular on *sketch* and *cartoon* datasets.

Task A

In this task we run a simple *grid-search* algorithm using nested for loops for different sets of hyperparameters. In particular for batch sizes of 128, 256 and 512 and learning rates of 0.005, 0.001 and 0.01 using always 5 epochs for time saving. It has been run the train on *photos* once and simultaneously they have been saved the best hyperparameters epoch by epoch for each combination, doing an average on the validation of *cartoon* and *sketch*. All the results can be found in Table 1.

LR / BATCH	0.005	0.001	0.01
128	32.5%	23.0%	**34.8**%
256	21.4%	20.4%	31.1%
512	31.1%	24.5%	25.1%

Table 1: Grid-search without DANN

Task B

Using the best set of hyperparameters found so far (batch size of 128 and learning rate of 0.01) the performances on *art_painting* reach 45% of accuracy.

Task C

In this task we run a simple *grid-search* algorithm using nested for loops for different sets of hyperparameters. In particular for batch sizes of 128, 256 and 512, learning rates of 0.005, 0.001 and 0.01 and alphas of 0.03 and 0.1 using always 5 epochs for time saving. It has been run the train on *photos* once and simultaneously they have been saved the best hyperparameters epoch by epoch for each combination, doing an average on the validation of *cartoon* and *sketch*. All the results can be found in Table 2. In general we can see an average improvement of about 5% with respect to the standard AlexNet with peaks of 10%.

BATCH	ALPHA	LR: 0.005	LR: 0.001	LR: 0.01
128	0.03	32.8%	22.7%	41.3%
	0.1	**43.8**%	26.0%	29.2%
256	0.03	24.9%	21.1%	32.2%
	0.1	31.5%	25.0%	38.8%
512	0.03	25.6%	30.8%	27.0%
	0.1	23.1%	27.7%	32.4%

Table 2: Grid-search with DANN

Task D

Using the best set of hyperparameters found so far (batch size of 128 and learning rate of 0.01 with alpha 0.1) the performances on *art_painting* reach 52% of accuracy, an improvement of about 9% with respect to the standard AlexNet.