ID: s267647
Name: Giorgio
Surname: Maritano

Homework 3 – Machine Learning and Artificial Intelligence

Implementing the model

For this homework I needed to implement a DANN (Domain Adversarial Neural Network) architecture starting from Alexnet's source code. Basically I use the standard fully connected layers of Alexnet as the DANN label predictor, while the feature extractor is implemented with Alexnet standard convolutional layers.
Whereas for what concerns the domain classifier I take the fully connected layers and change the last layer in a way which makes it have just 2 output neurons.

```
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, 2),
)
```

After this step I had to make sure that, if Alexnet is loaded with pre-trained weights, those weights also apply to the new branch which identifies as the domain classifier: so I had to copy them to the new classifier..
To make this happen I edited the *alexnet* function this way in the *if pretrained* block:

```
model = AlexNet(**kwargs)
if pretrained:
    state_dict = load_state_dict_from_url(model_urls['alexnet'],
                                          progress=progress)
    model.load_state_dict(state_dict, strict=False)
    model.dann_classifier[1].weight.data = model.classifier[1].weight.data
    model.dann_classifier[1].bias.data = model.classifier[1].bias.data
return model
```

Now my aim is to implement the *Gradient Reversal Layer*: this layer stays between the Feature Extractor and the Domain Classifier. Conceptually this layer's aim is to make samples from the two domains we analyse indistinguishable for the classifier.
In this homework the domains which I will use are Photo for training and Art and Painting for test.

This layer has two different behaviours whether there is a forward pass or a back propagation: if there is a *forward pass* it is basically an identity function whereas it multiplies the output for -1 during the *backpropagation* which will lead to the opposite of gradient descent.
This is done because the goal of the feature extractor is to minimize classification loss of the label predictor and maximize classification loss of domain predictor.

For the implementation of the new forward function I relied on the template I was provided by the teachers and added the needed lines of code.

```
# DANN forward function
def forward(self, x, alpha=None):
  features = self.features(x)

  # If we pass alpha, we can assume we are training the discriminator
  if alpha is not None:
    # flatten
    features = features.view(-1, 256 * 6 * 6)
    # gradient reversal layer (backward gradients will be reversed)
    reverse_feature = ReverseLayerF.apply(features, alpha)
    discriminator_output = torch.flatten(reverse_feature, 1)
    discriminator_output = self.dann_classifier(discriminator_output)
    return discriminator_output

    # If we don't pass alpha, we assume we are training with supervision
  else:
    # standard forward function
    class_outputs = self.avgpool(features)
    class_outputs = torch.flatten(class_outputs, 1)
    class_outputs = self.classifier(class_outputs)
    return class_outputs
```

Domain Adaptation

After editing Alexnet source code as asked in the previous point now it is time to modify also the training function to adapt it to the DANN: to do so I loaded two different Dataset of the two domain taken into consideration and from these two I created three dataloaders, two from the Art&Painting dataset of which one is for evaluation and the other for training purposes and the last one is for training based on the Photo dataset.

After that I simply divided the data from the labels and executed the following lines, which simply are the implementation in the training phase of the DANN:

```
# Create tensors labels for Steps 2 and 3 with labels rispectively equal to 0 and 1
z = torch.zeros(src_lab.shape, dtype=torch.long).to(DEVICE)
o = torch.ones(tgt_lab.shape, dtype=torch.long).to(DEVICE)

# Set network in training mode
net.train()

optimizer.zero_grad() #Zeroing gradients

### STEP 1 ###
# Train Label predictor on source labels
out1 = net(src_img)
loss1 = criterion(out1, src_lab)

loss1.backward()

### STEP 2 ###
# Train discriminator on source data (all labels to 0)
out2 = net(src_img, alpha)
loss2 = criterion(out2, z)

loss2.backward()

### STEP 3 ###
# Train discriminator on target data (all labels to 1)
out3 = net(tgt_img, alpha)
loss3 = criterion(out3, o)

loss3.backward()
```

Now it’s time to start actually observe the results.

To begin with I train the network as it is on the Photo dataset and test it on the Art&Painting set, without domain adaptation: the accuracy I got was almost 50%.

Test Accuracy: 0.49169921875

After that first run I will edit the training phase in order to implement the actual DANN architecture.

Without changing any hyperparameter I initially set the value of *alpha* to 0.1 and observed how the accuracy changes:

Test Accuracy: 0.48193359375

We can observe that there are no such improvements respect to the previous result.

So I tried to change in other ways the value of alpha and the learning rate. After various attempts I found some values which quite improve the accuracy score that are the following:

Learning rate of 1e-2
Number of Epochs equal to 20 (because I noticed that the various losses were not improving after this epoch)
Alpha of 0.03

Those values led me to

Test Accuracy: 0.52136354062

Comparing the results we can observe that with a finer choice of hyperparameters the DANN goes a bit higher in terms of accuracy on the test set respect to the standard Alexnet, even if the difference is not that relevant.

To look for a better accuracy we have to move on the the last point and perform Cross Domain Validation.

Cross Domain Validation

This final point consists in performing hyperparameter optimization with and without DANN adaption by tuning hyperparameters during the training phase on the Photo domain and then testing it on the Cartoon and Sketch domains. Once these steps are done I have to average the results obtained and with the best parameters implement the *Domain Adaption* point again.

I decided to manually perform the tuning by using a series of nested loops which iterates over various values of hyperparameters trying all possible combinations.

The hyperparameters I am going to optimize are: *batch size, learning rate and alpha*. Also I kept 10 as number of epochs for the validation phase.

For steps 4.A and 4.B I must not use alpha and what I'm going to do is train my network on the Photo dataset, then validate it on the Cartoon and Sketch datasets. I interpreted the task as follows: perform a validation phase sequentially on Cartoon dataset and then Sketch dataset. After that see how the network performed and then average the best results for each validation phase and confront them with the other results obtained with other combination of hyperparameters, but remember to reset the network before starting again with each combination of hyperparameters.

In the end I must pick the hyperparameters combination which gave me the best result on the average of the two dataset and use it ho test on the Art and Painting Dataset.

The best hyperparameters I got from the validation phase without domain adaption are a batch size of 128 and a learning rate of 0.01, with a total average accuracy of 0.33044591846573607.

After that I simply used the train and test phases that were already implemented in the notebook to run the step 3A with these hyperparameters.

The results I got are the following: 0.49755859375 which basically is the same as the ones without cross-domain validation.

Now let’s move on the last point of the homework which consists in doing cross-validation **with** domain adaption.

To implement this point I followed the logic behind the cross validation without domain adaption, but of course I took the training phase I implemented in point 3B and I used it first with the Cartoon dataset as target dataset, performing the cross validation later and then I implemented a new training phase but this time with the Sketch dataset as the target dataset.

After that I did the second validation phase on the Sketch dataset and at this point I have all the results: this means I can average them to find out which are the best hyperparameters in this case and finally implement training and test phases as I did before but with these ones.

In this case we can observe that we are getting quite higher results on average accuracies during the validation phase: indeed if previously the highest was 0.33 now lots of combinations of hyperparameters exceeds 0.33 and goes up almost to 0.5 on 8 epochs.

The best average accuracy I got was with a batch size of 64, learning rate of 0.01 and alpha 0.05. With these values I get an accuracy of 0.48876953125.

Actually I expected the network to perform better than this, seen how previously the accuracy went up from 0.33 to 0.49, but I think that to achieve that is not sufficient to implement the point with the best hyperparameters but the network the best performed should be deepcopied to a best_network and then with this best_network should be carried out the test phase.