POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Corso di Laurea Magistrale in Ingegneria Informatica

Machine Learning and Artificial Intelligence

Homework 3: Deep Domain Adaptation

Student:

Pietro Basci

266004

ACADEMIC YEAR 2019/2020

Introduction

The purpose of this homework is to implement a Domain Adversarial Neural Network (DANN) using AlexNet to classify images belonging to different domains. The dataset considered is the PACS dataset, which consists of 4 different domains: *Photo*, *Art painting*, *Cartoon* and *Sketch* which so far considers the largest domain shift as it is from the different image style depictions. The training of the model has been performed on Google Colab that provides a free Jupyter notebook environment running entirely in the cloud and powerful computing resources. In deep learning powerful GPUs dramatically reduces the time required by the training phase and Google Colab offers Nvidia Tesla K80 GPU having 2496 CUDA cores and 12GB of VRAM.

Dataset

As introduced above, the dataset considered is PACS which is composed of *Photo*, *Art painting*, *Cartoon* and *Sketch* domains each of which consists of 7 classes. In order to create the Pytorch datasets, the *ImageFolder* class has been used. The code to do that is reported below.

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

PHOTO_DIR = 'Homework3-PACS/PACS/photo'
ART_DIR = 'Homework3-PACS/PACS/art_painting'
CARTOON_DIR = 'Homework3-PACS/PACS/cartoon'
SKETCH_DIR = 'Homework3-PACS/PACS/sketch'

# Prepare Pytorch train/test Datasets
photo_dataset = torchvision.datasets.ImageFolder(PHOTO_DIR, transform=train_transform)
art_dataset = torchvision.datasets.ImageFolder(ART_DIR, transform=eval_transform)
cartoon_dataset = torchvision.datasets.ImageFolder(CARTOON_DIR, transform=eval_transform)
sketch_dataset = torchvision.datasets.ImageFolder(SKETCH_DIR, transform=eval_transform)

# Check dataset sizes
print('Photo Dataset: {}'.format(len(photo_dataset)))
print('Art Dataset: {}'.format(len(art_dataset)))
print('Cartoon Dataset: {}'.format(len(cartoon_dataset)))
print('Sketch Dataset: {}'.format(len(sketch_dataset)))
```

Output:

```
Photo Dataset: 1670
Art Dataset: 2048
Cartoon Dataset: 2344
Sketch Dataset: 3929
```

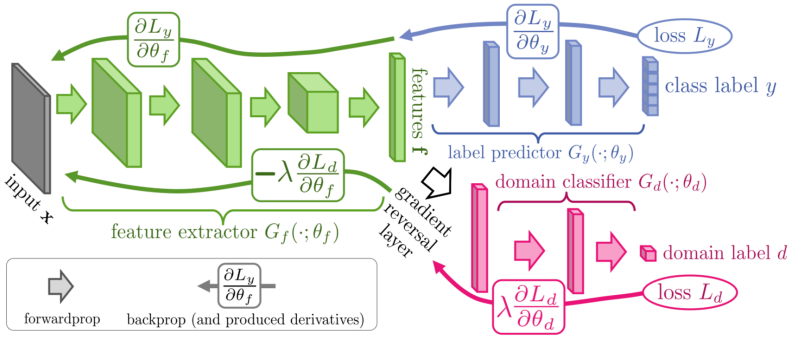
After that, Dataloaders are defined.

```
# Dataloaders iterate over pytorch datasets and transparently provide useful functions (e.g. parallelization and shuffling)
photo_dataloader = DataLoader(photo_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4, drop_last=True)
art_dataloader = DataLoader(art_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
cartoon_dataloader = DataLoader(cartoon_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
sketch_dataloader = DataLoader(sketch_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
```

The model

The Domain Adaptation is a technique that aims to improve performance in problems where training and test data come from similar but different distributions. The idea behind the model used is that, in order to achieve good performance on new distribution, predictions must be made based on features that cannot discriminate between the training (source) and test (target) domains. The model used is Domain Adversarial Neural Network (DANN) built using AlexNet. The architecture includes a features extractor Gf which is equal to the AlexNet convolutional layers, a label predictor Gy which corresponds to the AlexNet fully connected layers (so up to this point it is a standard AlexNet) and a domain classifier Gd which has the same structure of the AlexNet fully connected layers and is connected to the feature extractor via a gradient reversal layer that multiplies the gradient by a certain negative constant during the

backpropagation. Doing that, the trining proceeds minimizing the label prediction loss (for source example) and maximizing the domain classification loss (for all samples). Gradient reversal ensures that the feature distributions over the two domains are made similar (as indistinguishable as possible for the domain classifier), thus resulting in the domain-invariant features. The resulting network's structure is illustrated in the following image.



This type of network, as the training progresses, promotes the emergence of features that are (i) discriminative for the main learning task on the source domain and (ii) indiscriminate with respect to the shift between the domains. As a result the training phase produce a model that do not contains discriminative information about the origin of the input (source or target) and so it can generalize well from one domain to another. To define this model, the init function of AlexNet has been modified by adding a new classifier *dann_classifier* identical to the standard AlexNet *classifier*.

```
def __init__(self, num_classes=1000):
    super(AlexNet, self).__init__()
    self.features = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(64, 192, kernel_size=5, padding=2),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Conv2d(192, 384, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(384, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(256, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=3, stride=2),
    )
    self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
    self.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, num_classes),
    )
    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, num_classes),
    )
```

After that, both original network and the new classifier are initialized with the weights of ImageNet (at this point both classifiers have the same weights).

```
    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
        model.dann_classifier[4].weight.data = model.classifier[4].weight.data
        model.dann_classifier[4].bias.data = model.classifier[4].bias.data
        model.dann_classifier[6].weight.data = model.classifier[6].weight.data
        model.dann_classifier[6].bias.data = model.classifier[6].bias.data
```

Finally, also the forward function has been changed so that the alpha parameter indicate if a batch of data must go to label predictor or domain classifier. Specifically, if alpha is defined data goes to domain classifier and the gradient has to be reverted with the Gradient Reversal Layer, otherwise data goes to label classifier.

```
def forward(self, x, alpha=None):
    x = self.features(x)
    x = self.avgpool(x)
    x = torch.flatten(x, 1)

    if alpha is not None:
        # gradient reversal layer (backward gradients will be reversed)
        reverse_feature = ReverseLayerF.apply(x, 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:
        classifier_output = self.classifier(x)
        return classifier_output
```

Gradients are reverted by ReverseLayerF class whose backward method multiplies gradients by a negative constant.

```
class ReverseLayerF(Function):
    # Forwards identity
    # Sends backward reversed gradients
    @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 the model defining, since the AlexNet model has been developed for the ImageNet dataset which is composed by 1000 classes, the last layer of both fully connected layers (classifier and dann_classifier) have to be readapted to the number of classes of a domain (7) and the number of domains (2).

```
def prepare_net():
    net = alexnet(pretrained=True) # Loading AlexNet model

    # AlexNet has 1000 output neurons, corresponding to the 1000 ImageNet's classes
    # We need 7 outputs for PACS
    net.classifier[6] = nn.Linear(4096, NUM_CLASSES) # nn.Linear in pytorch is a fully connected layer
    # We need 2 outputs for Domains
    net.dann_classifier[6] = nn.Linear(4096, 2)

    return net
```

Then, the loss function, the optimizer and the scheduler are defined. In this phase, also the parameter to optimize are chosen: in this case, all the parameters of AlexNet will be optimized.

```
def prepare_train():
    # Define loss function
    criterion = nn.CrossEntropyLoss() # for classification, we use Cross Entropy

    # Choose parameters to optimize
    # To access a different set of parameters, you have to access submodules of AlexNet
    # (nn.Module objects, like AlexNet, implement the Composite Pattern)
    # e.g.: parameters of the fully connected layers: net.classifier.parameters()
    # e.g.: parameters of the convolutional layers: net.features.parameters()
    parameters_to_optimize = net.parameters() # In this case we optimize over all the parameters of AlexNet

    # Define optimizer
    # An optimizer updates the weights based on loss
    # We use SGD with momentum
    optimizer = optim.SGD(parameters_to_optimize, lr=LR, momentum=MOMENTUM, weight_decay=WEIGHT_DECAY)

    # Define scheduler
    # A scheduler dynamically changes learning rate
    # The most common schedule is the step(-down), which multiplies learning rate by gamma every STEP_SIZE epochs
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=STEP_SIZE, gamma=GAMMA)

    return criterion, optimizer, scheduler
```

In order to make a comparison between results obtained with and without domain adaptation, we first train a standard AlexNet without adaptation and then we compare results with those obtained using domain adaptation. We consider *Photo* as source domain and *Art painting* as target domain.

Training without Domain Adaptation

In this section, a standard AlexNet (without adaptation) has been trained. For each epoch, we iterate over all the dataset using more than one image at a time. In particular, at each iteration, the model is used to make prediction on a batch of n images (where n is defined by the *batch_size* parameter), then the average loss for those n samples is computed and finally weights are updated according to accumulated gradients. Since no alpha has been defined, data are sent only to label predictor and not to discriminator.

```
current_step = 0
# Start iterating over the epochs
for epoch in range(NUM_EPOCHS):
  print('Starting epoch {}/{}, LR = {}'.format(epoch+1, NUM_EPOCHS, scheduler.get_lr()))

  running_loss = 0.0
  # Iterate over the dataset
  for images, labels in train_dataloader:
    # Bring data over the device of choice
    images = images.to(DEVICE)
    labels = labels.to(DEVICE)

    net.train() # Sets module in training mode

    optimizer.zero_grad() # Zero-ing the gradients

    # Forward pass to the network
    outputs = net(images)
    # Compute loss based on output and ground truth
    loss = criterion(outputs, labels)
    running_loss += loss.item()

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

    # Compute gradients for each layer and update weights
    loss.backward()  # backward pass: computes gradients
    optimizer.step() # update weights based on accumulated gradients

    current_step += 1

  train_loss.append(running_loss/len(train_dataloader))  # compute average loss
  train_acc.append(get_accuracy(net, train_dataloader)) # compute accuracy on train set
```

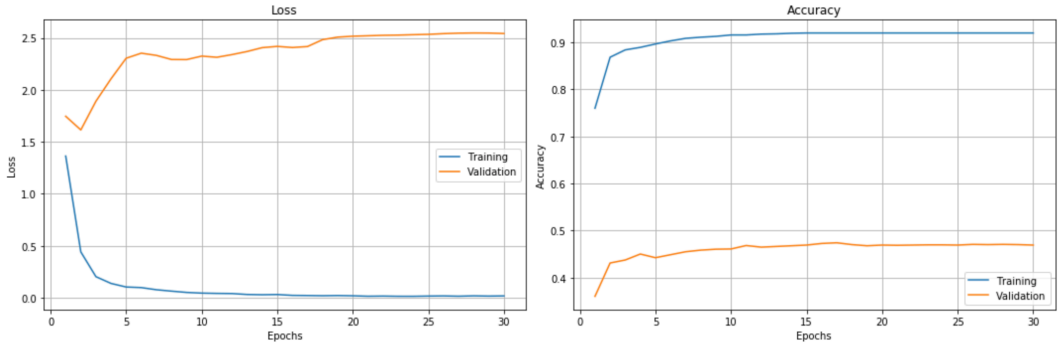
For each epoch, train loss and accuracy are traced to make a compare with validation loss and accuracy. At the end of each epoch, a test on the validation set has been performed in order to select the best performing model which will be used for the final evaluation on the test set.

```
# Evaluate the model on the validation set
net = net.to(DEVICE) # this will bring the network to GPU if DEVICE is cuda
net.train(False) # Set Network to evaluation mode

running_loss = 0.0
running_corrects = 0
for images, labels in val_dataloader:
  images = images.to(DEVICE)
  labels = labels.to(DEVICE)

  # Forward Pass
  outputs = net(images)

  # Compute loss based on output and ground truth
  loss = criterion(outputs, labels)
  running_loss += loss.item()

  # Get predictions
  _, preds = torch.max(outputs.data, 1)

  # Update Corrects
  running_corrects += torch.sum(preds == labels.data).data.item()

# Calculate Accuracy
accuracy = running_corrects / float(len(val_dataloader.dataset))

print('Test Accuracy on Validation: {}'.format(accuracy))

# Update the best model
if accuracy > best_accuracy:
  best_accuracy = accuracy
  best_net = copy.deepcopy(net)

val_acc.append(accuracy)
val_loss.append(running_loss/len(val_dataloader))

# Step the scheduler
scheduler.step()
```

By setting both *DOMAIN_ADAPTATION* and *VALIDATION* variables to False in the Main block, a training without adaptation on *Photo* dataset begins.

```
DOMAIN_ADAPTATION = False
VALIDATION = False

net = None

if not DOMAIN_ADAPTATION and not VALIDATION:
    net = prepare_net()
    criterion, optimizer, scheduler = prepare_train()
    # Train without domain adaptation on Photo
    net, _ = train(net, photo_dataloader)
```

Training using:
learning_rate = 1e-3; batch_size = 256; num_epochs=30; step_size=20.



Finally, the model has been tested on the test set.

```
# Test on art
accuracy = test(net, art_dataloader)
print('Test Accuracy: {}'.format(accuracy))
```

Output:

```
100%|██████████| 8/8 [00:04<00:00,  1.77it/s]Test Accuracy: 0.47412109375
```

Training with Domain Adaptation

In this section, the model with domain adaptation has been trained. Specifically, the network has been trained jointly on the labeled task (Photo) and the unsupervised task (discriminating between Photo and Art painting). The main difference with respect to the code of the previous section is that the training phase is divided into three steps: (i) train on source labels by forwarding source data Gy, get the loss and update gradients; (ii) train the discriminator by forwarding source data to Gd, get the loss (considering the label 0 for all data) and update gradients; (iii) train the discriminator by forwarding target data to Gd, get the loss (considering the label 1 for all data) and update gradients. Since in this case there are two dataset to iterate over, the *zip* function has been used to iterate over both dataset in parallel.

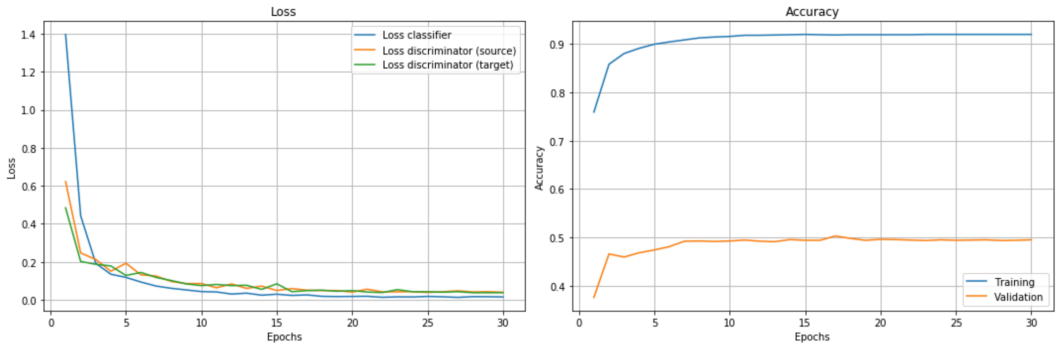
```
current_step = 0
# Start iterating over the epochs
for epoch in range(NUM_EPOCHS):
  print('Starting epoch {}/{}, LR = {}'.format(epoch+1, NUM_EPOCHS, scheduler.get_lr()))

  running_loss = 0.0
  running_loss0 = 0.0
  running_loss1 = 0.0
  # Iterate over the dataset
  for i, (source_data, target_data) in enumerate(zip(source_dataloader, target_dataloader)):
    source_images, source_labels = source_data
    target_images, _ = target_data

    # Bring data over the device of choice
    source_images = source_images.to(DEVICE)
    source_labels = source_labels.to(DEVICE)
    target_images = target_images.to(DEVICE)
```

```
net.train() # Sets module in training mode

optimizer.zero_grad() # Zero-ing the gradients

# Forward pass to the network
outputs = net(source_images)
# Compute loss based on output and ground truth
loss = criterion(outputs, source_labels)
running_loss += loss.item()
# Compute gradients for each layer and update weights
loss.backward()  # backward pass: computes gradients

# Forward pass to the network
outputs = net(source_images, alpha=ALPHA)
targets = torch.zeros(source_labels.size(0), dtype=torch.int64).to(DEVICE)
# Compute loss based on output and ground truth
discr_loss0 = criterion(outputs, targets)
running_loss0 += discr_loss0.item()
# Compute gradients for each layer and update weights
discr_loss0.backward()  # backward pass: computes gradients

# Forward pass to the network
outputs = net(target_images, alpha=ALPHA)
targets = torch.ones(source_labels.size(0), dtype=torch.int64).to(DEVICE)
# Compute loss based on output and ground truth
discr_loss1 = criterion(outputs, targets)
running_loss1 += discr_loss1.item()
# Compute gradients for each layer and update weights
discr_loss1.backward()  # backward pass: computes gradients
```

By setting *DOMAIN_ADAPTATION* variable to True and *VALIDATION* to False in the Main block, a training with adaptation on *Photo* dataset begins.

```
elif DOMAIN_ADAPTATION and not VALIDATION:
    net = prepare_net()
    criterion, optimizer, scheduler = prepare_train()
    # Train with domain adaptation on Photo to Art
    net, _ = train_dann(net, photo_dataloader, art_dataloader)
```

Training using:
learning_rate = 1e-3; batch_size = 256; num_epochs=30; step_size=20, alpha=0.01.



Finally, the model has been tested on the test set.

```
# Test on art
accuracy = test(net, art_dataloader)
print('Test Accuracy: {}'.format(accuracy))
```

Output:

```
100%|██████████| 8/8 [00:04<00:00,  1.69it/s]Test Accuracy: 0.5029296875
```

By using domain adaptation, we obtain an accuracy improvement between 1% and 2% (on average) depending on the random shuffle of data. The loss plot shows the trend of the 3 losses: the one on the labels and the two discriminator losses on source and target. As we can see, all curves goes near to 0 including the discriminator losses (which were expected to increase). By construction, as the model back-propagate a reversed gradient, by minimizing it, the feature extractor tries to maximize the error of the discriminator on source and target so losses should increase. However, the discriminator is still trying to minimize its own loss and due to its high number of neurons it is still able to recognize, among the informations extracted by the feature extractor, elements typical of the two domains (thanks also to the great difference between the two).

Cross Domain Validation

In order to find the best set of hyperparameters, we perform a cross domain validation: we run a grid search training on Photo and validating on Cartoon and Sketch, then we average accuracies and choose the set of hyperparameters with the highest average value of accuracy. This validation strategy has been used for both training with and without domain adaptation to compare results.

As first step, the model without domain adaptation has been trained and evaluated using this technique. Specifically, for each combination of hyperparameters (obtained using the cartesian product of the sets), two networks have been trained and evaluated on Cartoon and Sketch respectively. Accuracies of both models are saved in two lists whose values are then averaged and the resulting highest value in this new list is used to retrieve the best set of hyperparameters (by considering the position in the list). Due to limitations of Colab, the grid search has been performed with a limited number of combinations.

```
elif not DOMAIN_ADAPTATION and VALIDATION:
    # Train without domain adaptation on Photo to Cartoon and on Photo to Sketch and perform grid search
    LR_set = [5e-3, 1e-3, 1e-4]
    BS_set = [128, 256]

    hyperparams_sets=[]
    accuracies1 = []
    accuracies2 = []

    for i, hyperparams in enumerate(itertools.product(LR_set, BS_set)):
      LR, BATCH_SIZE = hyperparams

      net = prepare_net()
      criterion, optimizer, scheduler = prepare_train()
      print('(Photo to Catoon) trying with: LR={}, BS={}'.format(LR, BATCH_SIZE))
      # Train without domain adaptation on Photo to Cartoon
      _, accuracy = train(net, photo_dataloader, cartoon_dataloader)

      # Save hyperparameter set and accuracy of the net
      hyperparams_sets.append(hyperparams)
      accuracies1.append(accuracy)

      net = prepare_net()
      criterion, optimizer, scheduler = prepare_train()
      print('(Photo to Sketch) trying with: LR={}, BS={}'.format(LR, BATCH_SIZE))
      # Train without domain adaptation on Photo to Sketch
      _, accuracy = train(net, photo_dataloader, sketch_dataloader)

      # Save accuracy of the net
      accuracies2.append(accuracy)

    # Compute average of accuracies
    avg_accuracies = (np.array(accuracies1) + np.array(accuracies2)) / 2.0

    # Get the set of hyperparameter with the highest average accuracy
    LR, BATCH_SIZE = hyperparams_sets[list(avg_accuracies).index(np.array(avg_accuracies).max())]
    print('\n**Best hyperparamiters: LR={}, BS={}**\n'.format(LR, BATCH_SIZE))
```

The highest accuracies obtained (among all epochs) on Cartoon and Sketch and their averages are reported in the tables below.

Table 1 - Accuracies on Cartoon without Domain Adaptation

	BS=128	BS=256
LR=0.005	0.2986348122866894	0.3408703071672355
LR=0.001	0.25127986348122866	0.2785836177474403
LR=0.0001	0.23165529010238908	0.3361774744027304

Table 2 - Accuracies on Sketch without Domain Adaptation

	BS=128	BS=256
LR=0.005	0.3031305675744464	0.31712904046831253
LR=0.001	0.3018579791295495	0.23873759226266225
LR=0.0001	0.26851616187325017	0.21379485874268261

For both distributions, the best set of hyperparameters are LR=0,005 and BS=256.

Table 3 - Average accuracies without Domain Adaptation

	BS=128	BS=256
LR=0.005	0.30088269	0.32899967
LR=0.001	0.27656892	0.25866061
LR=0.0001	0.25008573	0.27498617

So also on the averages the best combination will be LR=0,005 and BS=256.

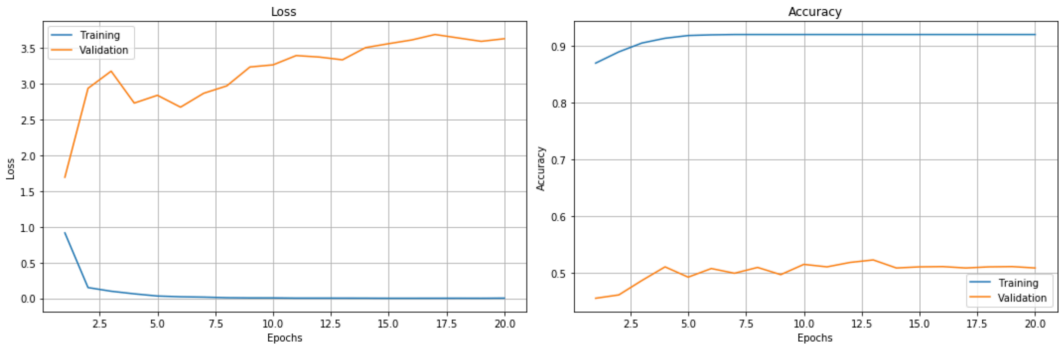
Output:

```
**Best hyperparamiters: LR=0.005, BS=256**
```

Once found the best hyperparameters, a new network has been trained on Photo using those parameters.

```
net = prepare_net()
criterion, optimizer, scheduler = prepare_train()
# Train a new net with the best hyperparamiters found
net, _ = train(net, photo_dataloader)
```

Training using:
learning_rate = 5e-3; batch_size = 256; num_epochs=20.



Finally, the model has been tested on Art painting.

```
# Test on art
accuracy = test(net, art_dataloader)
print('Test Accuracy: {}'.format(accuracy))
```

Output:

```
100%|██████████| 8/8 [00:05<00:00,  1.81it/s]Test Accuracy: 0.5234375
```

By only optimizing parameters, we obtained an accuracy improvement of about 5% with respect to the previous case.

In a similar way, the model with domain adaptation has been trained and evaluated using this technique. In this case, we consider also APHA as a parameter to optimize.

```
elif DOMAIN_ADAPTATION and VALIDATION:
    # Train with domain adaptation on photo to cartoon and on photo to sketch and perform grid search
    LR_set = [5e-3, 1e-3, 1e-4]
    BS_set = [128, 256]
    ALPHA_set = [0.03, 0.01, 0.001]

    hyperparams_sets=[]
    accuracies1 = []
    accuracies2 = []

    for i, hyperparams in enumerate(itertools.product(LR_set, BS_set, ALPHA_set)):
      LR, BATCH_SIZE, ALPHA = hyperparams

      net = prepare_net()
      criterion, optimizer, scheduler = prepare_train()
      print('(Photo to Catoon) trying with: LR={}, BS={}, ALPHA={}'.format(LR, BATCH_SIZE, ALPHA))
      # Train with domain adaptation on Photo to Cartoon
      _, accuracy = train_dann(net, photo_dataloader, cartoon_dataloader)

      # Save hyperparameter set and accuracy of the net
      hyperparams_sets.append(hyperparams)
      accuracies1.append(accuracy)

      net = prepare_net()
      criterion, optimizer, scheduler = prepare_train()
      print('(Photo to Sketch) trying with: LR={}, BS={}, ALPHA={}'.format(LR, BATCH_SIZE, ALPHA))
      # Train with domain adaptation on Photo to Sketch
      _, accuracy = train_dann(net, photo_dataloader, sketch_dataloader)

      # Save accuracy of the net
      accuracies2.append(accuracy)

    # Compute average of accuracies
    avg_accuracies = (np.array(accuracies1) + np.array(accuracies2)) / 2.0

    # Get the set of hyperparameter with the highest average accuracy
    LR, BATCH_SIZE, ALPHA = hyperparams_sets[list(avg_accuracies).index(np.array(avg_accuracies).max())]
    print('\n**Best hyperparamiters: LR={}, BS={}, ALPHA={}**\n'.format(LR, BATCH_SIZE, ALPHA))
```

The highest accuracies obtained (among all epochs) on Cartoon and Sketch and their averages are reported in the tables below.

Table 4 - Accuracies on Cartoon with Domain Adaptation

	ALPHA=0,03		ALPHA=0,01		ALPHA=0,001	
	BS=128	BS=256	BS=128	BS=256	BS=128	BS=256
LR=0,005	0.29180887372013653	0.35964163822525597	0.2999146757679181	0.27047781569965873	0.2879692832764505	0.30930034129692835
LR=0,001	0.21672354948805546	0.30076791808872372	0.31313993174061433	0.382679180887372	0.24829351535836178	0.302901023890785
LR=0,0001	0.27431740614334447	0.20563139931740615	0.2090443686006826	0.25170648464163825	0.2717576791808874	0.23378839590443687

Table 5 - Accuracies on Sketch with Domain Adaptation

	ALPHA=0,03		ALPHA=0,01		ALPHA=0,001	
	BS=128	BS=256	BS=128	BS=256	BS=128	BS=256
LR=0,005	0.28378722321201132	0.40264698396538856	0.2924408246373123	0.3318910664291168	0.2766607279205905	0.30109442606261133
LR=0,001	0.24306439297531118	0.25808093662509546	0.3054212267752609	0.34410791550012726	0.31254772206668363	0.28175108170017815
LR=0,0001	0.21175871723084755	0.1980147620259608	0.2967676253499618	0.2868414354797658	0.21659455332145583	0.19470603206922882

The best set of hyperparameters on Cartoon is LR=0,001, BS=256, ALPHA=0,01 which allows to achieve an accuracy of 0,38 on Cartoon. The best set on Sketch instead is LR=0,005, BS=256, ALPHA=0,03 which allows to achieve an accuracy 0,40 on Sketch. In both cases we have a significant improvement with the use of domain adaptation (+4% on Cartoon and +8% on Sketch with respect to the case without domain adaptation).

Table 6 - Average accuracies with Domain Adaptation

	ALPHA=0,03		ALPHA=0,01		ALPHA=0,001	
	BS=128	BS=256	BS=128	BS=256	BS=128	BS=256
LR=0,005	0.28779805	0.38114431	0.29617775	0.30118444	0.28231501	0.30519738
LR=0,001	0.22989397	0.27942443	0.30928058	0.36339355	0.28042062	0.29232605
LR=0,0001	0.24303806	0.20182308	0.252906	0.26927396	0.24417612	0.21424721

The best set considering the averages is LR=0,005, BS=256, ALPHA=0,03 with an average accuracy of 0.38 (+6% with respect to the case without domain adaptation).

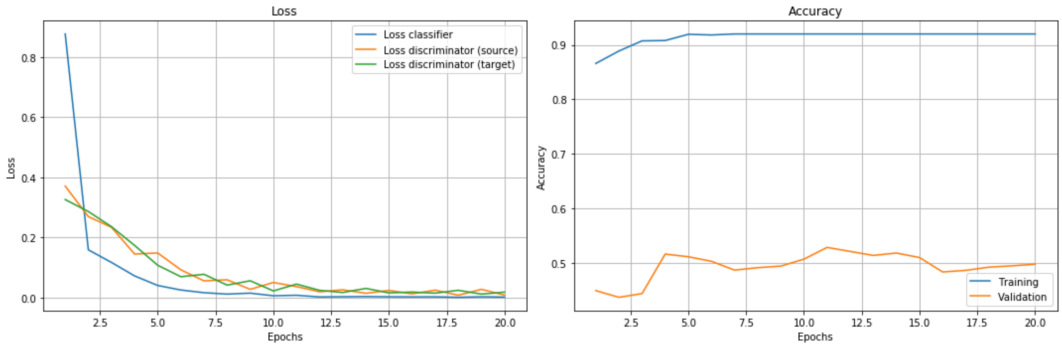
Output:

```
**Best hyperparamiters: LR=0.005, BS=256, ALPHA=0.03**
```

Once found the best hyperparameters, a new network has been trained on Photo using those parameters.

```
net = prepare_net()
criterion, optimizer, scheduler = prepare_train()
# Train a new net with the best hyperparamiters found
net, _ = train(net, photo_dataloader)
```

Training using:
learning_rate = 5e-3; batch_size = 256; num_epochs=20, alpha=0,03.



Finally, the model has been tested on Art painting.

```
# Test on art
accuracy = test(net, art_dataloader)
print('Test Accuracy: {}'.format(accuracy))
```

Output:

```
100%|██████████| 8/8 [00:05<00:00,  1.76it/s]Test Accuracy: 0.5380859375
```

By optimizing parameters, we obtained an accuracy improvement of about 4% with respect to the previous case. However, despite there are important improvements with the use of domain adaptation on *Cartoon* and *Sketch* domains (+4% and +8% respectively), the accuracy improvement on *Art painting* still remains on 1,5-2% with respect to the case without domain adaptation.