ID: s267647
Name: Giorgio
Surname: Maritano

Homework 2 – Machine Learning and Artificial Intelligence

1 – Data Preparation

In this section what I was required to do was to edit the code I was given to make it load images from the provided train / test splits present in the Github repo and to use the provided Caltech class (used as Dataset) to store them and to filter out the BACKGROUND class, which must not be used, implementing all the logic.

I made the Caltech class the most general possible, which allowed me to use it without further edit also in the next point with the validation set.

So in the __init__ function after receiving all the parameters the class reads the file provided as “split” and saves all the entries in a list.

After that it execs the following lines:

```
# Filter out the lines which start with 'BACKGROUND_Google' as asked in the homework
self.elements = [i for i in lines if not i.startswith('BACKGROUND_Google')]

# Delete BACKGROUND_Google class from dataset labels
self.classes = sorted(os.listdir(os.path.join(self.root, "")))
self.classes.remove("BACKGROUND_Google")
```

This way I completed the first point.

Of course I also implemented the Caltech’s methods to return the length of the dataset (by simply returning self.elements’ length) and to iterate over the dataset, using the pil_loader function provided to pass the data as a PIL image.

2 – Training from scratch

Here I was asked to split the original training set into a new training set and a validation set, of the same size

Since the code is widely commented I will just add a screenshot of this code snippet to show how I’ve done it.

I chose to execute this part outside of the Caltech class, because this way I could pass just the paths to the new files as the parameters and the class behaves just as it did before.

```
# 2 - Training from scratch
# Open the train.txt file and instantiate two lists
training = open(SPLIT_TRAIN, "r")
buffer = training.readlines()
buffer_class = []

training.close()

# Add each class of the buffer's corresponding entry to the buffer_class list
# In this way the element in buffer[i] will be of class buffer_class[i]
for ent in buffer:
  cl = ent.rstrip().split('/')[0]
  buffer_class.append(cl)

# I used the train_test_split method with the option stratify to have same percentage of examples of each class
# both in train and validation dataset
train, validation, y_tr, y_val = train_test_split(buffer, buffer_class, test_size=0.5, stratify=buffer_class)

# Here i simply create two files and write down the new entry and validation splits.
t = open("Homework2-Caltech101/train_2.txt", "w+")
v = open("Homework2-Caltech101/validation_2.txt", "w+")

for e in train:
  t.write(e)

for e in validation:
  v.write(e)

t.close()
v.close()

TRAIN2 = 'Homework2-Caltech101/train_2.txt'
VALID2 = 'Homework2-Caltech101/validation_2.txt'

# Now from here I can create two datasets for the train2 and val2
train2 = Caltech(DATA_DIR, split = TRAIN2, transform=train_transform)
valid2 = Caltech(DATA_DIR, split = VALID2, transform=eval_transform)
```

Part B of this section asks me to perform a model selection on the validation set after each training epoch and then choose the best one and perform the test phase with this model.

To determine the best one I relied on the accuracy value, saving epoch after epoch the best one (if it was better than the current) and also doing a deepcopy of the net network.

```
  ### Validation phase ###
  net.train(False)

  running_corrects = 0
  for images_v, labels_v in EDL2:
    images_v = images_v.to(DEVICE)
    labels_v = labels_v.to(DEVICE)

    # Forward Pass
    outputs = net(images_v)

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

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

  # Calculate Accuracy
  accuracy = running_corrects / float(len(valid2))
  print("\nAccuracy = {}\n".format(accuracy))

  if accuracy > best_acc:
    bestnet = copy.deepcopy(net)
    best_acc = accuracy

  # Step the scheduler to change the LR
  scheduler.step()

print("\n\n\n best net is: ")
print(bestnet)
```

To effectively check the results I executed two different test phases with the two nets:

100%|██████████| 12/12 [00:10<00:00, 1.14it/s]
Test Normal Net Accuracy: 0.09194607673695127

100%|██████████| 12/12 [00:09<00:00, 1.20it/s]

Test Best Net Accuracy: 0.1652264085724162

The results above are also the result obtained with the default hyperparameters we were given.

Now I changed some of them and I will report the results here:

- Learning Rate changes from 0.001 to 0.01

100%|██████████| 12/12 [00:09<00:00, 1.33it/s]
Test Best Net Accuracy: 0.2865537504320774

- Batch size = 512, so I need also to double the learning rate which will become 0.02. I also will increase the number of epoch from 30 to 50 and the step size from 20 to 30

100%|██████████| 6/6 [00:09<00:00, 1.51s/it]
Test Best Net Accuracy: 0.3259592118907708

As we can see there is a significant increase in accuracy from the network with 0.001 as LR and the one with 0.01, whereas the increase is less significant when we double it doubling also the batch size.

3 – Transfer Learning

Here I was asked to load a Alexnet model which was already trained on the ImageNet dataset (a really huge dataset compared to the Caltech101) which has the weights values related to that training phase.

Of course I had to change the Normalize function in the data preprocessing, using appropriate values for the mean and standard deviation. Here it is how I did it:

```
# Imagenet transforms
img_transform = transforms.Compose([transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.485,
0.456, 0.406), (0.229, 0.224, 0.225))
])
```

while I loaded the network with the instruction:

```
ptalexnet = alexnet(pretrained=True)
```

I run the first experiment with the default values, and the results are really impressive compared to the ones we got without a pre-trained network. At the end of the training phase I already had a loss value of 0.01993320882320404, while in the previous cases it used to be around at least 3.xx or 4.xx.

The accuracy after running the Test phase with the best pretrained network instead (I kept the validation phase inside the training loop) was:

Test Best Pretrained Net Accuracy: 0.8299343242309022

which is really high.

Now I'm going to post here some result with different sets of hyperparameters

- Batch size of 512 and learning rate of 2e-3 are the parameters changed from the default ones:

Test Best Pretrained Net Accuracy: 0.832353957829243

For the next experiment I will keep the changed values of batch size and learning rate and decrease the number of epochs and the step down policy respectively to 20 and 13.

- Batch size of 512, learning rate of 2e-3, 20 epochs and 13 as the step down policy:

Test Best Pretrained Net Accuracy: 0.8244037331489803

This third run instead will see the number of epochs increased to 50, but the learning rate and batch size will be halved, so their values will be again the default ones.

- Batch size of 256, learning rate of 1e-3, 50 epochs and 30 as the step down policy:

Test Best Pretrained Net Accuracy: 0.8313169720013827

In the end I will try to reset to default all values and experiment with just a very high learning rate of 0.6 just to see if the model will diverge or not:

- Learning rate of 0.6:

In this case there is no need to go through all the training phase and the test one, since already at the first epoch we get this result:

Starting epoch 1/30, LR = [0.6]
Step 0, Loss 4.850302696228027
Step 10, Loss ***nan***

With higher learning rate values of course we get higher losses, but if we choose a value which is **too** high the loss will diverge, as we can see from this test.

We can observe that we got the best accuracy with the default batch size and learning rate, and that with 30 or 50 epoch the result is almost the same.

For the last point of this section about transfer learning I first trained the network with the hyperparameters which got me the best accuracy from the previous point (pre – trained on the Imagenet Dataset) by optimizing only the parameters of the fully connected layers, and I got the following accuracy on the test set: *0.8282060145178016* which is slightly lower than the best accuracy I got before. What changed the most was the loss: now *0.17769354581832886* instead of *0.01993320882320404*.

Instead if we train only the convolutional layers of the same network with the same sets of hyperparameters we get an accuracy of *0.3183546491531282* and a loss of *3.308408737182617*, which are the worst parameters so far.

As we can see optimizing some layers' parameters instead of optimizing every parameters can cause the accuracy to drop down a lot if we freeze the fully connected layers and train only on the convolutional layers, whereas the accuracy for the network with only the fully connected layers' parameters optimized is very similar to the original one, even if with an higher loss (and that's logical because if we optimize just a part of the network it may be faster during the training phase but it for sure cannot be as good as a network fully trained).

4 – Data Augmentation

Since the most training data, the better, I applied other preprocessing functions to the training data only with the objective of having a better accuracy than before. Here I will report three preprocessing functions I applied on data with the relative results:

- RandomHorizontalFlip: this function has by default a 0.5 probability of flipping horizontally a PIL image. I observed no consistent variations in the final accuracy on the test set or the value of the loss at the end of the training phase compared to the results of the network without data augmentation.

- RandomErasing (applied to Tensor): with this kind of transformation instead we got some improvements in the final accuracy on the test set. For example, considering the initial network which had an accuracy of 9% more or less on the test set, with this kind of transformation applied to its training dataset we managed to increase it to *0.15935015554787418*.

- RandomRotation: this function also improved the test set accuracy, even if in a slightly weaker way than the previous one: in fact the accuracy I got is *0.14759764949879017*.

Now I'm going to apply a combination of those functions for data augmentation and see the results: I chose the RandomChoice transform and gave it the two previous functions which are not applied to tensors and it will randomly apply one of those. Plus I also kept the RandomErasing function which gave me the best results.

With this combination I got ad accuracy on the test set of 0.09194607673695127, so there aren't particular improvement in the accuracy.

I also decided to apply some of these transformation to the pretrained Alexnet, to see if the behaviours and the improvements were the same: both of them showed almost no accuracy improvement for the pretrained Alexnet network.