ROB313 TP2 : Siamese Neural Networks for beginners

Zheyi SHEN & Dajing GU

Décembre 2020

Exercise 1 : Mnist classification with Pytorch

Q1 - Explanation of the script

The file `script_pytorch_simple.py` given by professor defines a neural network. Outside the loop, the code first creates random inputs and outputs and initializes the weights. In the loop, the forward propagation is defined, which includes three layers. Then, the loss function is created according to which the backward propagation is defined in order to update the weights, so that during the training, the model can be optimized.

Q2 - A script using the Module class

In this question, we write a network which has the same architecture as the one in the file `script_pytorch_simple.py`. According to the code, we can see that there are three layers, two of which are linear transformation layers and the remaining one is ReLU layer. The code can be found in TP2 - EX1 .

The size of the picture in the dataset Mnist is 28×28. The but is to determine the number in the picture which is in 0 to 9, so the input size is 784 and the output size is 10. We use `nn.CrossEntropyLoss()` as the loss function and apply the backward loop with `loss.backward()`.

Q3 - Add summaries

We can use `SummaryWrite` in tensorboard to keep the training loss and training accuracy. Two another parameters we should keep are learning rate and batch size because they can easily influence the performance of CNN model.

Q4 - Convolution neural network Resnet18

Now, we transform the network to a convolutional neural network Resnet18. The size of an MNIST image is $28 \times 28 \times 1$ and that of an ImageNet is $224 \times 224 \times 3$. So we change the channel number of the funtion `self.pre` to 1 in class ResNet and resize the MINST images to shape (224,224) in order to make the MNIST image and ResNet18 adapt to each other.

Q5 - Hyper-parameter optimization

It is important to find good parameters because the neural networks might have totally new results with new parameters. The Table 1 and Table 2 show the accuracy with different learning rate, weight decay and batch size. Because the training is slow, we only record the accuracy after 5 epochs.

weight decay \ λ	1.0	0.1	0.01	0.001
0.1	9.80%	10.09%	13.23%	98.25%
0.01	9.82%	14.95%	98.34%	99.21%
0.001	60.72%	97.63%	99.33%	99.21%

Table 1: Weight decay/learning rate (λ) table with batch size = 64

batchsize \ λ	1.0	0.1	0.01	0.001
5	9.74%	8.92%	98.83%	99.47%
64	60.72%	97.63%	99.33%	99.21%
200	73.07%	98.61%	99.34%	98.98%

Table 2: Batchsize/learning rate (λ) table with Weight decay = 0.001

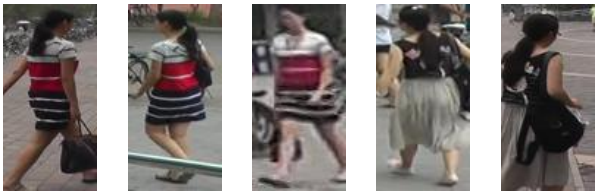
Q6 - Regularisation

The regularization present in our network is the weight decay. To avoid over-fitting, we need to add a regularization term to loss function. The basic idea is to reduce the influence of unimportant parameters on the final result and the useful weights in the network will not be affected by weight decay. `weight decay` is a coefficient of this term.

Exercise 2 : Re identification

In this section, we focus on the the performance of Siamese networks on Market-1501 database, a ReID dataset. It contains 32668 images of 1501 persons. Code associated can be found here: TP2 - EX2 .

Q1 - Have a look at the dataset



(a) (b) (c) (d) (e)

Figure 1: Illustration of the dataset

The function `plot_images.py` is implemented to illustrate the dataset. We find that the it is composed of photos of pedestrians taken from various cameras, with each photo of shape $[128 \times 64 \times 3]$.

Q2 - Implementation of the triplet loss

Instead of learning to classify its inputs, Siamese networks learn the similarity between 2 inputs, for which the triplet loss is used. The triplet loss is smaller for two similar images but huge for different images. The code associated can be found in the link of this exercise.

Q3 - Description of pre-implemented `RandomSampler.py`

In the given code, `RandomSampler` is called in the initialization of the class `Data`, more precisely in the function `self.train_loader`.

In order to start the training, we need to offer the model with the images in the dataset. `Sampler` is capable of giving us the corresponding indexes of the images needed, according to which `Data_loader` will then obtain the corresponding images from the dataset. `RandomSampler`, as one of the samplers, is capable of shuffling the indexes.

There are three main methods in `RandomSampler`:

- `__init__` : Initialization
- `__iter__` : Generation of index values specifying the data to be read in each step.
- `__len__` : Length of each iterator.

Q4 - Completement of the `data.py`

The two methods needed to be implemented are presented here.

```
def __getitem__(self, index):
    img = Image.open(self.imgs[index-1]) #3*64*128
    target = self._id2label[self.get_id(self.imgs[index-1])]
    if self.transform is not None:
        img = self.transform(img)
    return img, target

def __len__(self):
    return len(self.imgs)
```

Q5 - Create our own DNN

A Resnet 50 is used as our own DNN, whose last layer has been replaced by two fully connected layers, a part of the codes associated are presented below, the whole codes can be found in the link.

```
def __init__(self):
    # write the CNN initialization
```

```
    super(REID_NET, self).__init__()
    model = resnet50(pretrained=True)
    self.fc_hidden1 = 1024
    self.fc_hidden2 = 768
    self.resnet=torch.nn.Sequential(*(list(model.children())[:-1]))
    # delete the last FC layer of resnet

    fc_features = model.fc.in_features

    self.fc_id = nn.Sequential(
                    nn.Linear(fc_features, self.fc_hidden1),
                    nn.Linear(self.fc_hidden1, num_classes)
                    )
    self.fc_metric = nn.Sequential(
                    nn.Linear(fc_features, self.fc_hidden1),
                    nn.Linear(self.fc_hidden1, num_classes)
                    )
```

In order to improve the training, a classification loss is used as auxiliary loss. In `loss.py`, the triplet loss and the cross-entropy loss are combined. The input of the triplet loss is `predict_id` and the labels, while the input of the cross-entropy loss is `predict_metric` and the labels. The formula of the final loss is:

$$Loss_{sum} = Loss_{triplet} + 2 \times Loss_{CrossEntropy}$$

Q6 - Implementation of the training code

The implemented function `train()` is presented below.

```
def train(model,train_loader,scheduler,optimizer,loss_function):
    model.train()
    for (inputs, targets) in train_loader:
        inputs, targets = inputs.to('cuda'), targets.to('cuda')
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_function(outputs, targets)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

The function `evaluate()` has calculated (CMC) the Cumulative Matching Characteristics curves and mAP (mean average precsion) to evaluate the performance the model developed on ReID.

It first extracts features from the dataset *query* and *test*. The CMC and mAP are then calculated in using thie features extracted.

Q7 - Saving and loading the model

The code is already given by the professor, which is shown in colab.