Aaron Riedling
Christoph Hoffmann

1	Σ

Assignment Sheet 5

Optimising the neural network

We built our neural network based on the code skeleton supplied. Three different types of optimisations were tested.

Network architecture

We used a Convolutional Neural Network (CNN) with two different parts. One was based on a 5x5 kernel, one on a 9x9 kernel. After two layers each we added a dropout layer and then combined the results and fed it into fully connected layers. As activation function we used leaky ReLU. For one of the systems, the conv layers were reduced to 3 into 6 and 6 into 12 layers (1a/1b and 2a/2b, respectively).

```
class Net(nn.Module):
    def __init__(self, dropout):
        super(Net, self).__init__()
        self.conv1a = nn.Conv2d(3, 12, kernel_size=5)
        # 3 input images: RGB
        self.conv2a = nn.Conv2d(12, 24, kernel_size=5)
        # 2 conv layers for feature extraction
        self.conv_DOa = nn.Dropout2d(dropout)
        self.conv1b = nn.Conv2d(3, 12, kernel_size=7)
        # 3 input images: RGB
        self.conv2b = nn.Conv2d(12, 24, kernel_size=7)
        # 2 conv layers for feature extraction
        self.conv_DOb = nn.Dropout2d(dropout)
        self.fc1 = nn.Linear(24*24*45 + 24*14*28, 512)
        self.fc2 = nn.Linear(512, 128)
        #self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 1)

    def forward(self, x):
        # a-arm of CNN --> smaller features
        xa = F.max_pool2d(self.conv1a(x),2)
        xa = F.leaky_relu(xa,0.05)
        xa = self.conv2a(xa)
        xa = self.conv_DOa(xa)
        xa = F.max_pool2d(xa , 2)
        xa = F.leaky_relu(xa , 0.05) # x.size -> BS*24*24*45
        #print(xa.size())
        #exit()
        xa = xa.view(-1,24*24*45) # 1 neuron per subimage

        # b-arm of CNN --> larger features
        xb = F.max_pool2d(self.conv1b(x) , 3)
        xb = F.leaky_relu(xb,0.05)
        xb = self.conv2b(xb)
        xb = self.conv_DOb(xb)
        xb = F.max_pool2d(xb , 2)
        xb = F.leaky_relu(xb , 0.05) # x.size -> BS*24*10*19
        #print(xb.size())
        #exit()
        xb = xb.view(-1,24*14*28) # 1 neuron per subimage

        # combine
        x = torch.cat((xa, xb),1)
        x = F.leaky_relu(self.fc1(x), 0.05)
        x = F.leaky_relu(self.fc2(x), 0.05)
        #x = F.leaky_relu(self.fc3(x), 0.05)
        output = torch.sigmoid(self.fc4(x))
        return output
```

Train image transformations

To have more data for training we did two things. First we created grayscale versions of all images, and second we added random transformations. We tested RandomRotate, RandomPerspective and some others, but in the end just stuck with RandomVerticalFlip and RandomHorizontalFlip. This doubled the number of images available for train and for each images randomly added three flipped versions. In total, this multiplies the number of image variants by a factor of 8. Importantly, this is only done for the training data. No image manipulation is done on the test images since we did not want to train our network how to work on modified images. It thus was important to not use the network with the lowest loss, but with the best test accuracy, which the code skeleton automatically did.

```
gs_image = transforms.Grayscale(3)(orig_image)
# three channels are needed to match the input dimensions
gs_image.save(train_path + "GS_" + file)
label_list["GS_" + file] = Yval
# Yval is the same value as the original, label_list is the read-in JSON

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomVerticalFlip(0.5),
    # This normalization is used on the test server
    transforms.Normalize([0.2404, 0.2967, 0.3563], [0.0547, 0.0527, 0.0477])
    ])
```

Parameter fine-tuning

Lastly we implemented some options to fine-tune parameters including Momentum, Learning Rate Decay, Dropout Chance. We switched the SGD optimiser for an Adam optimisier and from MSE loss to BCE loss. We also used train loss as a tiebreaker when two models had the same test accuracy. We also implemented a flag for random train-test splitting at different ratios. A typical call for the tool is:

```
python3 ./AR_CH_Boat.py
            -k 5
            --nsplits 1
            --lr 0.001
            --lr-redux 0.00005
            --momentum 0.5
            --dropout 0.3
            --epochs 30
            --inflate-train
```

This command runs the training using the standard data split in the downloaded data, the standard learning rate of 0.1, but with a learning rate decrease of 0.005 after each epoch that starts after 3 additional epochs of full learning rate. Momentum is 0.5 and dropout is 0.3. *–inflate-train* induces creation of the grayscale images which doubles the amount of train data.
Importantly, this uses a hard-coded input path. An argument -i can be used to set an alternative path. At that path the code expects a folder called ”original_split” which contains the original unpacked data with the json file directly in the folder and two subfolders train and val. This structure is the source of training data which will not be modified, but copied and then modified.
After training and validation, these copied data are not removed and can be reused by the argument *–reuse-split*. Note that *–reuse-split* in combination with *–inflate-train* can lead to repetitive inflation which is undesired.

Evironment

The environment was setup with the following commands. No GPU was used since the laptop had none.

```
conda create -n boat_p36 python=3.6
conda activate boat_p36
pip install tqdm
conda install pytorch torchvision torchaudio cpuonly -c pytorch
conda install matplotlib
```