Convolutional Neural Network for Binary Classification

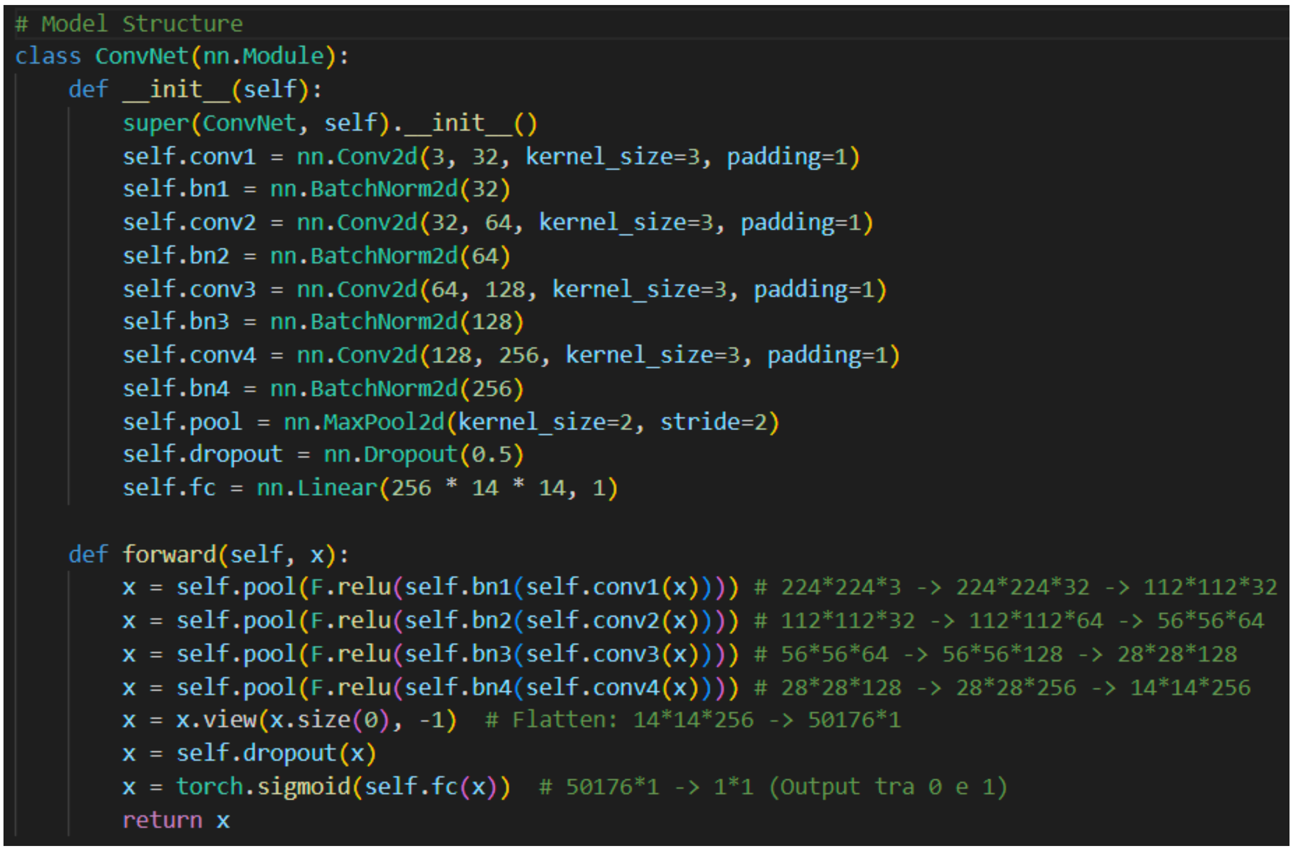
By: Francesco Galassi, Livio Marzio della Penna

THE TASK

The task of this challenge was to correctly determine if, given an input image, there was some kind of fire depicted in it.

THE NETWORK STRUCTURE

To do this, we decided to follow a classic Binary Classification approach, based on the use of a simple, self-made CNN, built in the following way:



```python
# Model Structure
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(256)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(256 * 14 * 14, 1)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x)))) # 224*224*3 -> 224*224*32 -> 112*112*32
        x = self.pool(F.relu(self.bn2(self.conv2(x)))) # 112*112*32 -> 112*112*64 -> 56*56*64
        x = self.pool(F.relu(self.bn3(self.conv3(x)))) # 56*56*64 -> 56*56*128 -> 28*28*128
        x = self.pool(F.relu(self.bn4(self.conv4(x)))) # 28*28*128 -> 28*28*256 -> 14*14*256
        x = x.view(x.size(0), -1)  # Flatten: 14*14*256 -> 50176*1
        x = self.dropout(x)
        x = torch.sigmoid(self.fc(x))  # 50176*1 -> 1*1 (Output tra 0 e 1)
        return x
```

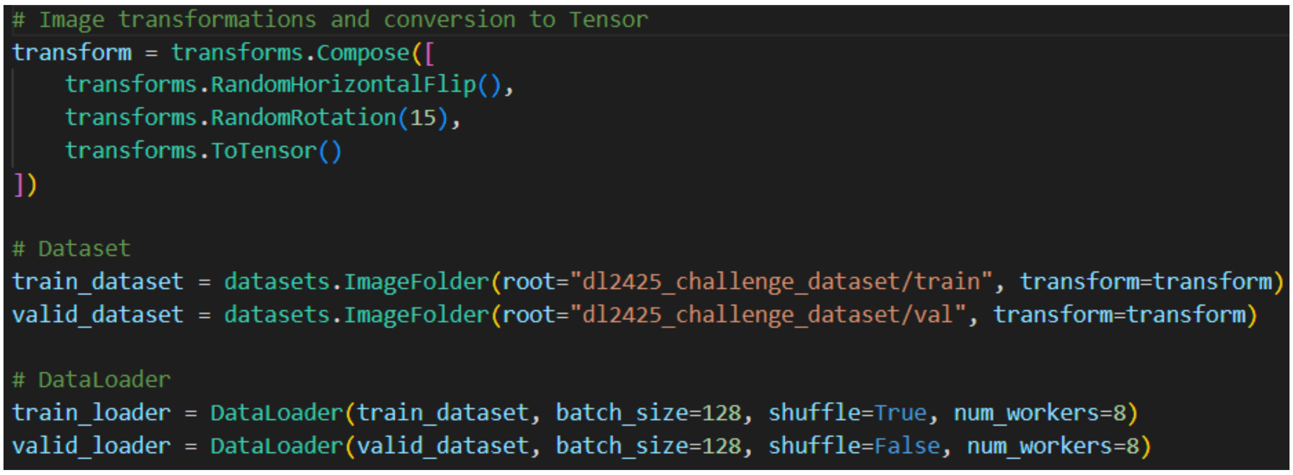
1. **Features Extraction Layers**: we used a sequence of convolutional layers, separated from each other by a ReLU activation function (for non-linearity) and a max-pooling.
   At first we tried to use different kernel sizes, however we noticed that - because of the 'simplicity' of the task, and also considering the enormous image size (HxW) reduction, derived from the use of one separated MaxPool layer for each Convolutional one - using a standard 3x3 kernel was the optimal choice.
   By also applying a padding of 1, we made sure that the dimensions of the feature map were the same as the original image's.
   As per usual, after each convolution we make sure to normalize the resulting tensor with a BatchNorm layer, which acts as a regularization factor for our model.

We used a total of 4 sequences of Conv-ReLU-MaxPool. We could've used more of such layers, however - considering both our hardware resources and the fact that each Convolutional layer is non-linearly separated (and thus each convolution matrix will have its weights updated separately, giving a more precise final result) - we decided that 4 were enough to obtain a good compromise between accuracy and training time.

2. **Classification Layers**: after we extract our features matrix, we flatten it into a 1-dimensional vector.
   This vector is then passed as input to our Fully-Connected (linear) layer, which returns a single value. This number is then transformed by a sigmoid function, to output the probability of the image containing fire.
   During this process, to avoid a biased learning given by the presence of overly used features, we also inserted a Dropout layer with dropping probability p = 0.5.

THE DATASET

In order to maximize the efficiency of the model's training, we made sure to apply some prior transformations to our images.



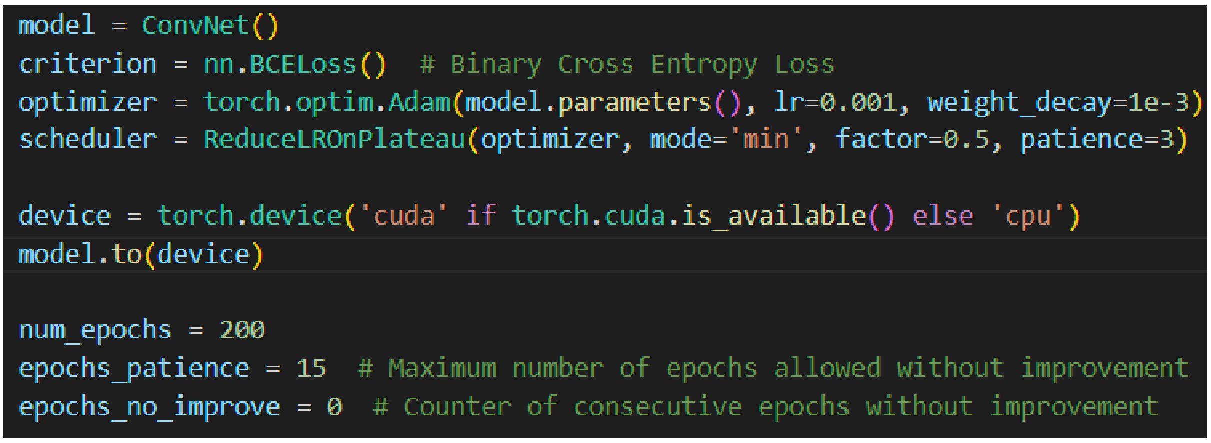
```
# Image transformations and conversion to Tensor
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ToTensor()
])

# Dataset
train_dataset = datasets.ImageFolder(root="dl2425_challenge_dataset/train", transform=transform)
valid_dataset = datasets.ImageFolder(root="dl2425_challenge_dataset/val", transform=transform)

# DataLoader
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True, num_workers=8)
valid_loader = DataLoader(valid_dataset, batch_size=128, shuffle=False, num_workers=8)
```

1. First of all, we noticed that the dataset was already split into train, validation and test subsets, respectively occupying 70%, 20% and 10% of the whole dataset. Since we only had a total of approximately 15.000 colored images – each one in .jpeg format, with size 224x224 – we decided to maintain such proportions at training time.
   We then decided to proceed with batches of 128 images each (considering a suggested size of 50 to 100 images per batch) trying once again to balance training time with HW capabilities.

2. To avoid overfitting, we applied data augmentation: this will ensure that the model won't just simply 'memorize' the images but it will actually be able to classify a new unseen input.

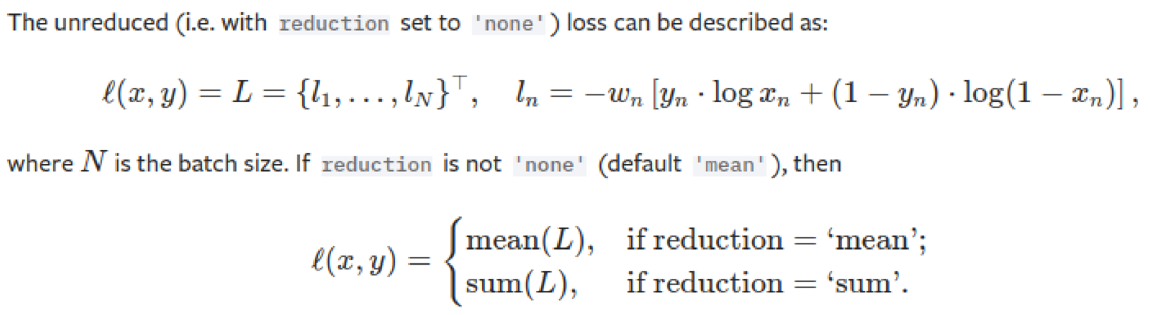
THE TRAINING

For training the model, while we were already sure on which Loss function and optimizer to use, we tried some different possible approaches on the usable hyper-parameters to give to these functions.



```
model = ConvNet()
criterion = nn.BCELoss()  # Binary Cross Entropy Loss
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-3)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=3)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)

num_epochs = 200
epochs_patience = 15  # Maximum number of epochs allowed without improvement
epochs_no_improve = 0  # Counter of consecutive epochs without improvement
```

1. First of all, we decided to use BCE, one of the best and most suggested loss functions when it comes to binary classification: the BCE expects a probability p as the classifier's output, which will then be used to compute the Cross Entropy for each predicted. The final result is thus given by the mean of all computed cross entropies of the batch:



The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

2. The computed loss is then backpropagated and used by the Adam optimizer to update the weights. We chose to use this algorithm instead of the SGD approach because of its simplicity to use and faster convergence.
   At first, we tried using a learning rate of 0.00005, however the use of batch-normalization allowed us to increase it by 20 times, reaching better results in way less epochs.
   We also tried using some few different weight decay values before actually deciding on 0.001, as it seemed a good tradeoff between generalization and possibility of overfitting.

3. One thing we wanted to do was to dynamically change the learning rate while training the model: we used a scheduler that, after 3 epochs of no improvement in the loss, automatically halves the learning rate of the optimizer.
   This helps especially when there are continuous oscillations in the loss (and by consequence, also in the accuracy), by drastically lowering them and thus learning more slowly.

4. Finally, to avoid wasting time and energy for useless computations with no benefits, we also implemented an early-stopper: after 15 epochs of no improvement in the accuracy, the model trained so far will be saved and the training will stop.
   The patience was chosen, by trying to always respect the rule of thumb of approximately 10% to 20% of the maximum number of epochs (200 in our case).

THE RESULTS

After approximately 1 hour of training (only with the final values and parameters), the model correctly classified 97.34% of the images of the validation set.

To classify the images from the test set, we loaded our best model obtained so far, which then inferred the label for each image and saved both the result on the .csv file.

We also made some tests changing our CNN with a pretrained ResNet18, from which we reset the last fully connected layer and trained the model again on our dataset, obtaining a 99% accuracy. However, in the end we wanted to try with our own skills, starting from zero and building our network with our hands. Especially, we didn't consider the idea of using the ResNet18 fair, since it's a model already trained for large multiclass classification on millions of images, as well as having a state-of-the-art structure (thanks mainly to its residual block) already built-in.

On our GitHub you can find:

1. Self-made CNN training script;
2. Self-made CNN training log;
3. Self-made CNN best trained model;
4. Self-made CNN results' csv creator script;
5. Pre-trained ResNet18 training script.