Deep Learning challenge binary classifier: fire or not?

Riccardo Inverardi Galli-2058055 and Domenico Menale-2049488

This report is intended as a short summary of our work of building a binary classifier for the Deep Learning class at ACSAI Sapienza. The classifier is trained to recognize the presence or the absence of fire in 224x224 RGB pictures taken from the provided dataset. No additional images where used in training the model, other than the ones provided by the professors, so that it could be as accurate as possible over the testing dataset distribution.

Training the model

Initial architecture. To tackle the task of image recognition, the most immediate approach is to use a Convolutional Neural Network, in order to take advantage of its multi-channel expansion abilities and low number of parameters. Our initial network (class MLP in MLP.py) consisted of four 3x3 convolutions with no stride nor padding with a reducing number of channels (16-8-4-1), followed by six linear layers which reduced the input to a 2D dimensionality (2809-512-128-64-32-16-2). The training was initially done on a laptop. The model showed an accuracy on the validation set that hit its maximum value at 88%, no matter the value of the training loss.

AlexNet Architecture. Once we switched training to Kaggle, we were able to train a network with more parameters. Our second model (class MLPAlex in MLP.py) is an implementation of AlexNet. This network performed much better than our custom architecture within the first few epochs of training. Although the model was trained for 60 epochs in total, in order to see if it was possible to further increase accuracy on the validation set, it was not possible to increase it over 96% (Fig. 1). It is important to note that this result was reached after just eleven epochs of training (as one can see in the Kaggle training log file). This fact, coupled with the very high accuracy value at epoch 0 (no training), suggests that the architecture is already capable of determining the presence or absence of fire with the proper initialization. When creating the model, we initialize the weights picking from a Xavier uniform distribution:

$$U \sim (-\sqrt{\frac{6}{n_{in}+n_{out}}}; \sqrt{\frac{6}{n_{in}+n_{out}}})$$

This approach has shown to be very effective in fighting vanishing and exploding gradients, compared to using a normal distribution. The model does not perform any pre-processing on the training samples, nor it does any data augmentation.

Training. Training is performed through the script train.py. The training set is loaded through torchvision.datasets.ImageFolder class, applying only a transformation that turns the images into tensors. Because of the initial low memory capabilities, we where training in epoch batches on only a subset of the training

dataset, choosing a random permutation of the indices. In order to provide the training samples to the network, we used

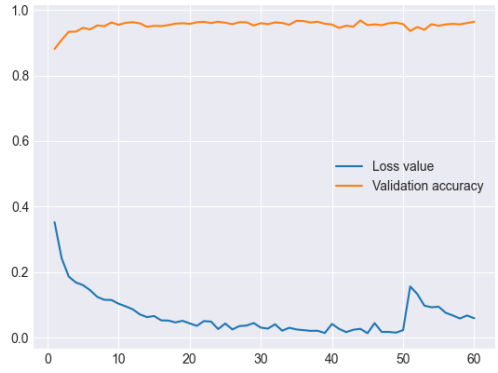

Fig. 1. Graph showing loss value over the training set and accuracy score on the validation set

torch.utils.data.DataLoader class, with a batch size of 64. The final model contains 46 million parameters. For the optimizer, we chose Adaptive Moment Estimation (ADAM) because it showed a faster converging behavior than Stochastic Gradient Descent (SGD). As hyperparameters, we set the learning rate to 0.5e-3 (0.0005), and chose to use L2 weight regularization through a weight decay hyperparameter set to 1e-2. As a loss function, we picked CrossEntropyLoss, that is widely used in classification tasks.

Training loop. The training loop (utils.py) is a standard loop, where we iterate through each epoch, generating an output of the model, computing the loss for each epoch, zeroing out the gradient before calling loss.backward(), and then performing the optimization step by calling optimizer.step().

Loss Value. The loss for each epoch is computed as the sum of the value of each batch divided by the total number of elements in the train dataset, as

$$\frac{1}{N}\sum_{i=1}^{N} L(x_i)$$

Validation. Validation accuracy is computed by iterating over the validation dataset, and then extracting the index of the highest value of the final layer through torch.max(outputs, dim=1). By using this method, we have no need of normalizing the output of the final layer through softmax. The same procedure is applied during the testing phase to extract the class to which each sample in the testing dataset belongs to.

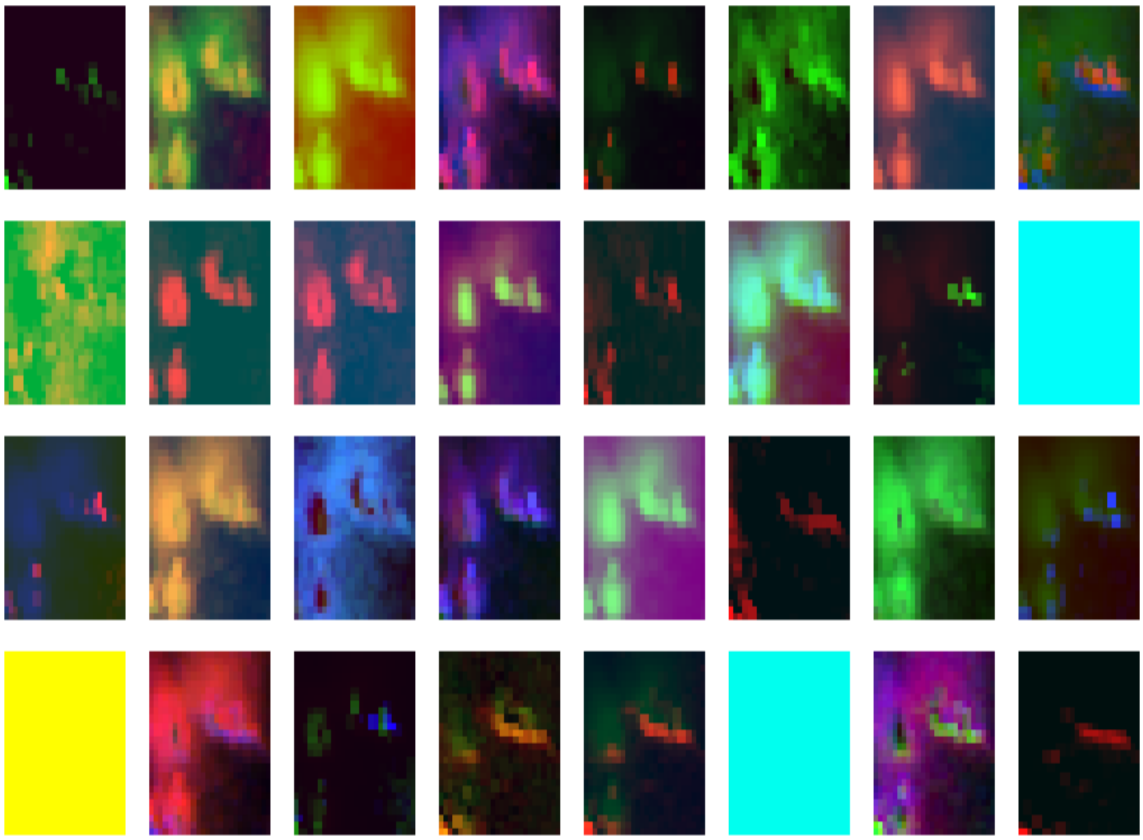

Fig. 2

Fig. 2. Output of the first convolutional layer