Project - CNN classifier

Computer Vision and Pattern Recognition

Marsich Gaia [SM3500600]

A.A. 2023 - 2024, Data Science and Scientific Computing

Repository with all the files: https://github.com/gmarsich/CVPR-project

Contents

1 Introduction

This project requires the implementation of an image classifier based on convolutional neural networks. The provided dataset (from [1.]), contains 15 categories (office, kitchen, living room, bedroom, store, industrial, tall building, inside city, street, highway, coast, open country, mountain, forest, suburb). Samples are shown in Fig. 1.

Figure 1: Examples of images from each of the 15 categories of the provided dataset

The images are in gray scale. The dataset was provided already divided into a train set, composed by 1500 images, and a test set, made of 2985 images.
The work, divided in three main tasks, has been carried out through Python.

2 Task 1

Firstly, I had to train a shallow network from scratch according to some specifications:

- **(1)** use the network layout shown in Fig. 2
- **(3)** resize the images to 64×64 so that the network can be properly fed
- **(4)** split the provided training set in 85% for actual training set and 15% to be used as validation set
- **(6)** employ the stochastic gradient descent with momentum optimization algorithm, using the default parameters of the library
- **(5)** use minibatches of size 32

- **(2)** set the initial bias values to 0 and use initial weights drawn from a Gaussian distribution having a mean of 0 and a standard deviation of 0.01
- **(7)** choose an appropriate learning rate and train for a few dozen epochs
- **(9)** report and discuss the plots of loss and accuracy during training, for both the training set and the validation set
- **(8)** comment on the criterion you choose for stopping the training
- **(10)** report the confusion matrix and the overall accuracy, both computed on the test set

#	type	size
1	Image Input	64×64×1 images
2	Convolution	8 3×3 convolutions with stride 1
3	ReLU	
4	Max Pooling	2×2 max pooling with stride 2
5	Convolution	16 3×3 convolutions with stride 1
6	ReLU	
7	Max Pooling	2×2 max pooling with stride 2
8	Convolution	32 3×3 convolutions with stride 1
9	ReLU	
10	Fully Connected	15
11	Softmax	softmax
12	Classification Output	crossentropyex

Figure 2: Layout of the CNN to be used for Task 1

The number in the parenthesis indicate the order I followed to tackle the assignments.

2.1 Approach and implementation choices

Following the indications provided by **(1)** and **(2)**, I created the class `CNN1` reported in the following snippet:

```
class CNN1(nn.Module):
    def __init__(self):
        super(CNN1, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3,
        ↪  stride=1)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3,
        ↪  stride=1)
```

```
10          self.relu2 = nn.ReLU()
11          self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)
12
13          self.conv3 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,
            ↪  stride=1)
14          self.relu3 = nn.ReLU()
15
16          self.fc = nn.Linear(32 * 12 * 12, 15)
17
18          # for (2) required by Task 1, see def initialize_weights(self)
19          self.initialize_weights()
20
21
22      def initialize_weights(self):
23          for module in self.modules(): # to recursively iterate over all the
            ↪  modules
24              if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
25                  init.normal_(module.weight, mean=0, std=0.01) # Gaussian
                    ↪  distribution: mean of 0, standard deviation of 0.01
26                  init.constant_(module.bias, 0) # set the bias to 0
27
28
29      def forward(self, x):
30          x = self.conv1(x)
31          x = self.relu1(x)
32          x = self.maxpool1(x)
33
34          x = self.conv2(x)
35          x = self.relu2(x)
36          x = self.maxpool2(x)
37
38          x = self.conv3(x)
39          x = self.relu3(x)
40
41          x = x.view(-1, 32 * 12 * 12)
42          x = self.fc(x)
43
44          return x
```

Only the most important comments have been reported in order not to make the text heavier. The structure of Fig. 2 has been followed. Some relevant remarks:

- in the first convolutional layer, at line 5, I put `in_channels=1`: the images are in gray scale, so there will be a unique channel
- the input of a layer had to be consistent with the output of the previous layer

- the `nn.Softmax()` layer was not inserted since, in the following, `nn.CrossEntropyLoss()` will be used (and this already incorporates a softmax operation)
- the point **(2)** was directly integrated into the class `CNN` through the method `initialize_weights`

The images were resized (**(3)**) during the loading of the images with `ImageFolder`: the parameter `transform` during this operation was set to the following method:

```
1  def transform(image):
2      resize = transforms.Compose([transforms.Resize([64,64]), # resize to 64x64
3                                   transforms.ToTensor(), # get a PyTorch tensor
4                                   transforms.Grayscale()]) #one-channel image
5      resized_image = resize(image)
6      resized_image = resized_image * 255.0 # revert the normalization
7      return resized_image
```

Observe that `transforms.ToTensor()` at line `3` does the transformation to a PyTorch tensor, but there is a side effect: the pixels' values will be scaled from the range [0, 255] to the range [0.0, 1.0], making images unrecognizable. A reversion of the normalization is necessary, and this is performed at line `6`.
The original training set was split according to the given instructions, so I got the actual training set with 1275 images and a validation set with 225 images (**(4)**).

Minibatches of size 32 were used (**(5)**) and the stochastic gradient descent with momentum optimization algorithm was implemented with `torch.optim.SGD`, setting `momentum=0.9` and using `nn.CrossEntropyLoss()` (**(6)**). The chosen learning rate was 0.0005 and the training has been performed for 48 epochs (**(7)**). The training phase is performed with the support of the previously mentioned validation set so that the generalisation capability of the network evaluated at a certain epoch can be estimated. The analysis keeps going until the number of epochs that was set is reached. The algorithm compares the model at the current epoch with the best model obtained up to that point (the performances are based on the validation loss). At the end, the more recent model will be the optimal one among those that were tested (**(8)**).

The method `train_model(model, EPOCHS, train_loader, val_loader, optimizer, loss_function)` was defined with the purpose of training the network. At each epoch, it prints the values of the training and validation losses, as well as the accuracy on the validation set. It returns `model_path`, `loss_train`, `loss_val`, `accuracies_train` and `accuracies_val`, respectively the path of the best model, the list containing the losses associated to the training, the list of the losses of the validation, the list with the accuracies on the training set and the list with the accuracies on the validation set.

2.2 Results

In the previous subsection I explained how everything was set up. The outputs of the `train_model(model, EPOCHS, train_loader, val_loader, optimizer, loss_function)` method were used to get the plots of loss (Fig. 3) and accuracy (Fig. 4) during training (**(9)**).

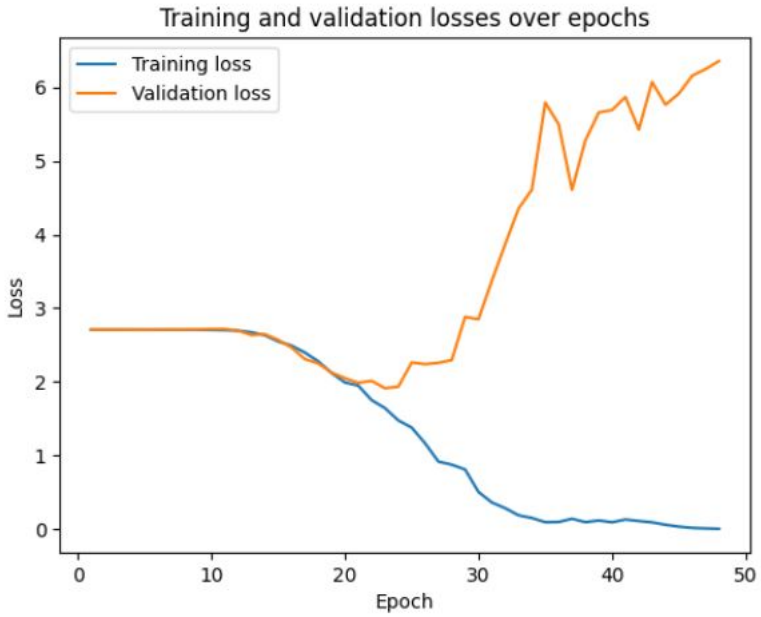

Figure 3: Plot representing the trend of training and validation losses over epochs

As expected, since the network is trained on a specific training set over and over, the loss associated to this set keeps decreasing. On the other hand, the tests done on a separate set, the validation set, report an increase of the loss after a certain point.

The plots of the accuracies reflect this concept as well: as a matter of fact, since the network got used to the specific training set, the accuracy on the same training set tends to reach the 100%. As one expects and wants from this procedure, also the accuracy on the validation set increases and its value settles around 30%.

Finally, using the test set on the trained model, I got an overall accuracy of 25%. The confusion matrix is reported in Fig. 5 (**(10)**).

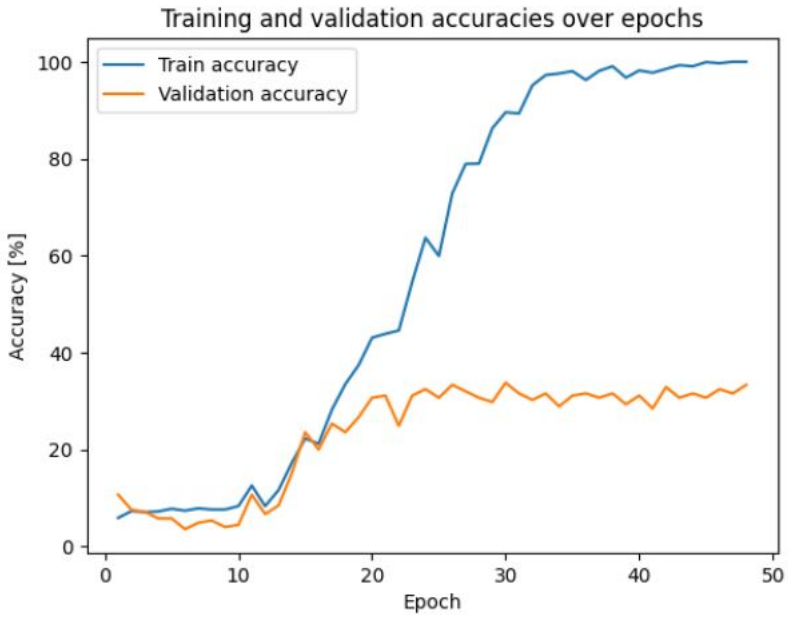

Figure 4: Plot representing the trend of training and validation accuracies over epochs

3 Task 2

In the previous section I obtained an accuracy of about 25%. In this second part the aim is to improve this result. Some suggestions are given by the text of the assignment:

- data augmentation: given the small training set, data augmentation is likely to improve the performance. For the problem at hand, left-to-right reflections are a reasonable augmentation technique
- batch normalization 3: add batch normalization layers before the reLU layers
- change the size and/or the number of the convolutional filters, for instance try increasing their support as we move from input to output: 3×3, 5×5, 7×7
- play with the optimization parameters (learning rate, weights initialization, weights regularization, minibatch size,...); you can also switch to the adam optimizer
- dropout: add some dropout layer to improve regularization
- employ an ensemble of networks (five to ten), trained independently. Use the arithmetic average of the outputs to assign the class, as in 4

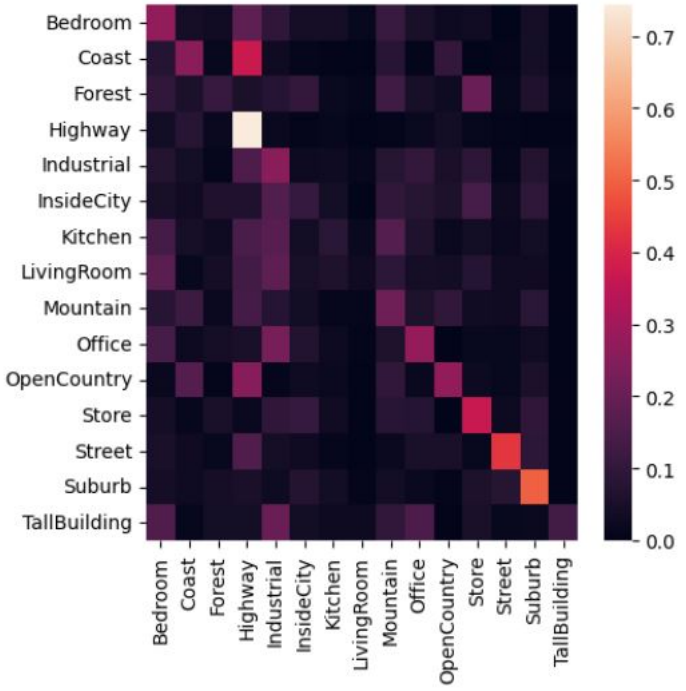

Figure 5: Confusion matrix computed on the test set, CNN1 network

I decided to implement the underlined propositions.

3.1 Approach and implementation choices

Two upgrades of the original `CNN1` class were prepared, here I summarize the differences:

- `CNN2`:
  - support size: the second convolutional layer has a 5x5 support, the third one has a 7x7 support
  - batch normalization: after each convolutional layer a batch normalization was added
- `CNN3`: upgrade of `CNN2` class, where a dropout (dropout probability of 0.4) was added right after the third ReLU

First of all, I applied a data augmentation: the only difference with the original version is the training dataset since "new" images are added to the initial set. The "new" images are simply the left-to-right reflections of the original training dataset.

In the trial with `CNN3`, instead of the stochastic gradient descent with momentum optimizer the Adam optimizer with learning rate equal to 0.0005 was used.

3.2 Results

Training a `CNN2` network using `nn.CrossEntropyLoss()` and a stochastic gradient descent with momentum optimizer, having learning rate equal to 0.0005 and `momentum=0.9`, I got an accuracy on the test set of 58% and the confusion matrix in Fig. 6.

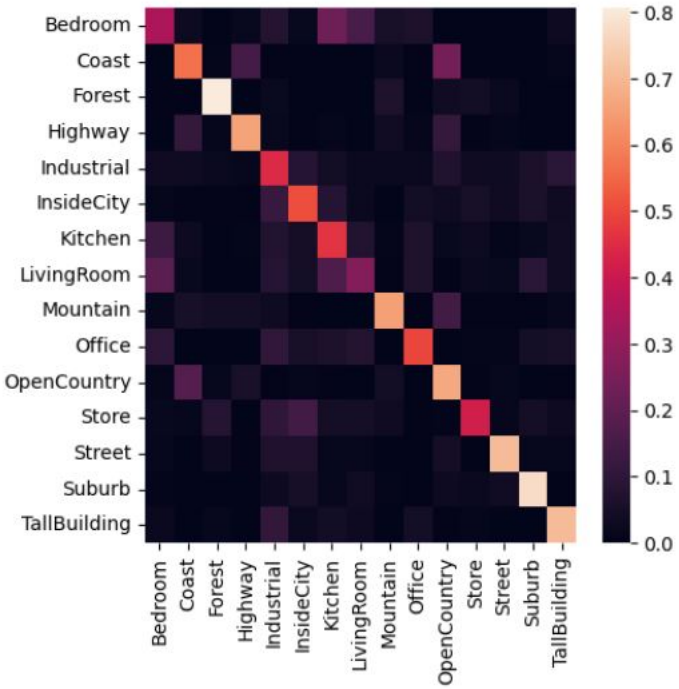

Figure 6: Confusion matrix computed on the test set, CNN2 network

An improvement in accuracy is evident compared with the base case of Task 1. Having a look at the two confusion matrices, in Fig. 5 and Fig. 6, one can notice that with `CNN2` the diagonal is better defined, has lighter colors (meaning that the results are better), and the off-diagonal elements are darker.
With the `CNN3` I obtained an accuracy on the test set of 56% and the confusion matrix in Fig. 7.
The results are slightly worse than the first improvement. However, the difference is small so more evidence would be needed to get a more certain feedback. It is possible, however, that there were no major improvements with the introduction of dropout and the use of Adam optimizer.

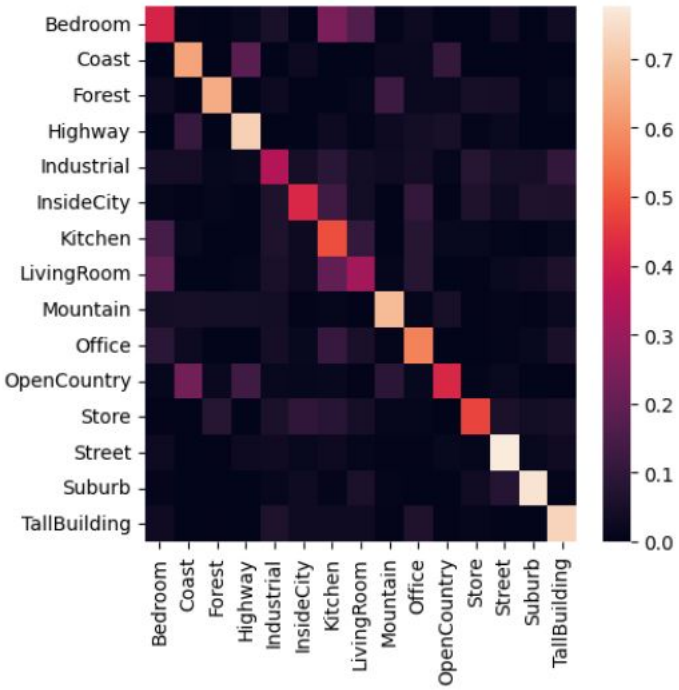

Figure 7: Confusion matrix computed on the test set, CNN3 network

4 Task 3

In the last task it is asked to use transfer learning based on a pre-trained network, for instance AlexNet 5, in the following two manners:

- **AlexNet (1)**: freeze the weights of all the layers but the last fully connected layer and fine-tune the weights of the last layer based on the same train and validation sets employed before
- **AlexNet (2)**: employ the pre-trained network as a feature extractor, accessing the activation of an intermediate layer (for instance, one of the fully connected layers) and train a multiclass linear SVM. For implementing the multiclass SVM use any of the approaches seen in the lectures, for instance DAG

4.1 Approach and implementation choices

A first important remark is that AlexNet requires a different type of input. First of all, 3 channels are needed; then, in the paper 5, it is written that the images should be 224x224. Additionally, the reversion of normalization in `transform(image)` should be avoided. So, I had to load the images using

`ImageFolder` and setting `transform` to `transform_ALEXNET`, which is defined as follows:

```
def transform_ALEXNET(image):
    resize = transforms.Compose([transforms.Resize([224,224]), # resize the
    ↪  images as indicated by the original paper
                                 transforms.ToTensor()])
    resized_image = resize(image)
    #resized_image = resized_image * 255. # if uncommented, the accuracy will
    ↪  decrease
    return resized_image
```

Again I did the division of the original training set into the actual training set and the validation set and I performed the data augmentation on the actual training set.

4.1.1 AlexNet (1)

The first proposed manner was implemented in this way:

```
alexnet = models.alexnet(weights=AlexNet_Weights.DEFAULT)

for param in alexnet.parameters(): # freeze all layers except the last fully
↪  connected layer
    param.requires_grad = False

in_features = alexnet.classifier[6].in_features # modify the last fully
↪  connected layer
alexnet.classifier[6] = nn.Linear(in_features, 15)

for param in alexnet.classifier[6].parameters(): # during training, only the
↪  weights and biases of the last layer will have their gradients updated
    param.requires_grad = True
```

Subsequently, I performed the training of the network using the stochastic gradient descent with momentum optimization algorithm and the `train_model` method previously mentioned.

4.1.2 AlexNet (2)

I defined the method `extract(model, loader)` to extract the features from a given loader (for my purpose, in particular I will need to use `model = alexnet`):

```
def extract(model, loader):
    features = []
```

```
    labels = []

    model.eval()
    with torch.no_grad():
        for inputs, target in loader:
            # Extract features from the intermediate layer
            interm_feat = model(inputs)
            features.append(interm_feat.view(interm_feat.size(0), -1).numpy())
            labels.append(target.numpy())

    # Concatenate features and labels
    features = np.concatenate(features, axis=0)
    labels = np.concatenate(labels, axis=0)

    return features, labels
```

Then, a multiclass linear SVM (exploiting the one-vs-rest scheme) was trained:

```
SVM = svm.LinearSVC(C=1.0, max_iter=1000) # uses one-vs-the-rest; C has the
↪  default value
SVM.fit(train_features, train_labels)
```

4.2 Results

The first approach gave an accuracy on the test set equal to 82% and the confusion matrix reported in Fig. 8.
The second approach gave an accuracy on the test set equal to 79.53%, computed with this piece of code:

```
y_pred = SVM.predict(test_features)
accuracy = accuracy_score(test_labels, y_pred)
print(f'Test Accuracy: {accuracy * 100:.2f}%')
```

5 More ideas

Something else that could have been done for further analysis:

- taking the exact time needed for the computations
- doing a table to compare different choices in the implementation (e.g., values of learning rates, `momentum`,...)
- doing separately the improvements suggested by the Task 2 and see how each of them performs. Then, maybe studying some other combinations

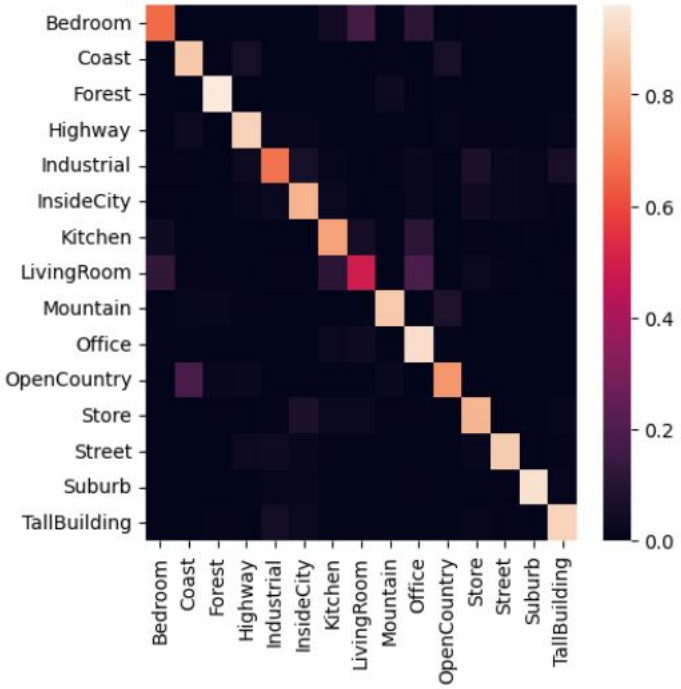

Figure 8: Confusion matrix computed on the test set, using the pre-trained network of AlexNet. The weights of all the layers but the last fully connected layer were frozen and the weights of the last layer were fine-tuned

6 References

1. Lazebnik, S., Schmid, C., and Ponce, J., *Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories*, in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2169-2178.

2. `https://dingyan89.medium.com/calculating-parameters-of-convolutional-and-fully-connected-layers-with-keras-186590df36c6`

3. Ioffe, S. and Szegedy, C., *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, *arXiv preprint arXiv:1502.03167*, 2015

4. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A., *Going deeper with convolutions*, in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015

5. Krizhevsky, A., Sutskever, I., and Hinton, G. E., *Imagenet classification with deep convolutional neural networks*, in *Advances in neural information processing systems*, pages 1097–1105, 2012