Venice Boats Classification

HW2 - ML Sapienza

Giorgio De Magistris

14 December 2018

1 Introduction

This report is about Venice boats Classification. The problem we're going to analyze consists into the assignment of correct "label" to a given snapshot of a Venice boat. We'll use techniques of supervised learning, meaning that we're given a dataset containing thousands of snapshots with the name of the class they belongs to, from which our models can learn. Formally our task is to learn an approximation $\hat{f}$ of the function $f : x \rightarrow t$ where x is a snapshot and t is the name of the class of x. The dataset is defined as a set of tuples $D = \{< x, t >^n\}$ where n is the cardinality . The vector x has three dimensions, $(height, width, channels)$, where height and width are expressed in pixels whose values vary in range $[0, 255]$ and we have three channels for each of the RGB components. The vector t has 23 deimensions, one for each of the different classes of boats, and all components will be zero except for the one representing the right class of the associated x. In this report we'll compare four different techniques, each of them will provide a different approximation $\hat{f}$ of the same function f.

2 Dataset

As we mentioned in the introduction we'll use supervised learning algorithms, and in order to make them work properly we need a lot of correct examples. To train our models we'll use MarDCT [1] dataset, that's already splitted into two parts, the first is for training and the second for testing. The trainingset contains 24 subfolders, 23 of them containing snapshots of each of the 23 classes of boats and the last one, named "Water", containing partial snapshots. The testset is structured in a single folder containing

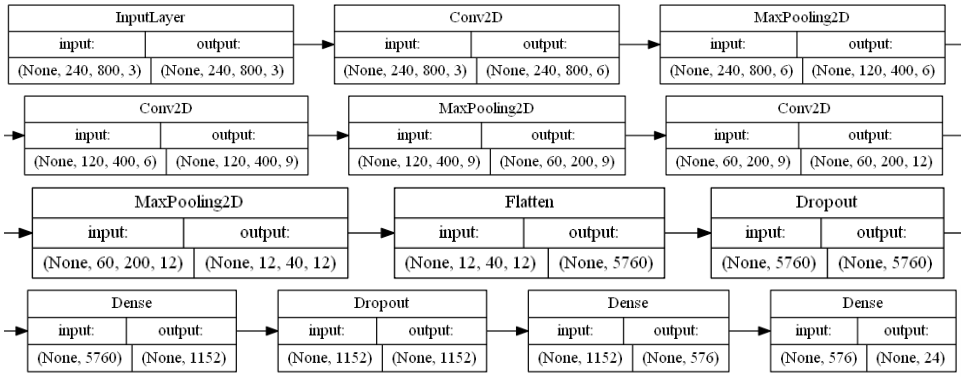


Figure 1: CNN structure

the snapshots and a txt file with the ground truth where each row has the format $image\_name; class\_name$. The trainingset contains 4774 snapshots balanced in the different classes while testset 1970, but some of them belongs to classes that are not in the trainingset, so after some filtering the remaining samples are fewer.

3 Convolutional Neural Network

3.1 Model Structure

The first approach we're going to use consists in defining from scratch a Convolutional Neural Network to be trained directly on the dataset. Figure 3 illustrate the structure, each node of the graph represents a layer, and for each layer there're two values for input and output shapes respectively. Note that the first dimension is the batch size and it's intentionally left blank, because it can be decided when the model is trained. The Input layer will receive a batch of snapshots with shape $(240, 800, 3)$. Convolutional layers convolute the input tensor $I(h_1 \times w_1)$ with a fixed size kernel $K(h_2 \times w_2)$ and the output tensor $O(h_1 \times w_1)$, the result of such operation, is defined as follows:

$$O[i,j] = \sum_{m=0}^{h1} \sum_{n=0}^{w1} I[i+m, j+n] K[m,n]$$

. If the input has more than one channel, as in our case, also the kernel will have more than one channel. Let's consider for example the first convolutional layer. The Input tensor has 3 channels, and since the convolutional

layer has 6 filters, the output will have 6 channels too. For each channel of the input we define 6 (2×2) kernels. Let's call I_i the i^{th} channel of the input, K_i^j the j^{th} kernel of the i^{th} channel and O_i the i^{th} channel of the output. We can compute the output as follows, $O_i = \sum_{j=1}^{channels}(I_i) \circledast (K_i^j)$. An important observation is that convolutional layers modify only the number of channels of the input. Pooling layers are used instead to reduce the dimension of the input, for example if we apply a max pooling layer with size $(h_s \times w_s)$ to an input image $I(h_i \times w_i)$ we're just applying a $(h_s \times w_s)$ grid to our image and we're building the output O taking only one element from each square of the grid, in this case the one with higher value, since we're applying MaxPooling. So, since squares do not overlap (we set strides equal to the pooling size) and since we chose the size such that in each level input width and height are divisible by that size, we can conclude that the output O will have shape $(\frac{h_i}{h_s} \times \frac{w_i}{w_s})$. So contrary to convolutional layers pooling layers reduce height and width of the input leaving the number of channels unchanged. Dense layers have one dimensional input and output, and each unit is connected to all units of the next level, so before feeding the dense layer with the output of the last pooling layer the last must be flattened, namely it must be transformed from a multidimensional layer $(height, width, filters)$ into one dimensional, with the new shape equal to $height \times width \times filters$. Now that we know the structure of the model we can sum up the total of trainable parameters. Each convolutional layer introduces $parameters = kernelsize \times filters \times inputchannels + biases$, pooling layers do not introduce parameters, since they compute a fixed operation, while the two piped dense layers introduce a number of parameters equal to the units of the first, times the units of the second, plus a number of biases that's equal to the number of units in the first layers that written in equation is $parameters = units_1 \times units_2 + units_2$. In table **??** we apply these formulas to compute the total number of parameters.

3.2 Performances

There're multiple reason why we implemented a CNN from scratch instead of using an existing famous model. The first attempt indeed was to train directly on MarDT dataset InceptionV3 [2], a convolutional neural with 23851784 parameters and 159 layers of depth and VGG19 [3], with 143667240 parameters and 26 layers. Even though these networks achieved excellent results on ImageNet[5] dataset, respectively top-1 accuracy of 94% and 90%, when trained on MarDT results were disappointing. This gap arises from the different sizes of the two datasets, one has more than 14

Layer	Kernel Size	Units	Parameters
Conv2D	$(5, 5)$		$5 \times 5 \times 6 \times 3 + 6 = 456$
MaxPooling			0
Conv2D	$(5, 5)$		$5 \times 5 \times 9 \times 6 + 9 = 1359$
MaxPooling			0
Conv2D	$(5, 5)$		$5 \times 5 \times 12 \times 9 + 12 = 2712$
Dense		5670	$5670 \times 1152 + 1152 = 6636672$
Dense		1152	$1152 \times 576 + 576 = 6641286636672$
Dense		576	$576 \times 24 + 24 = 13848$
Total			7319175

Table 1: CNN parameters

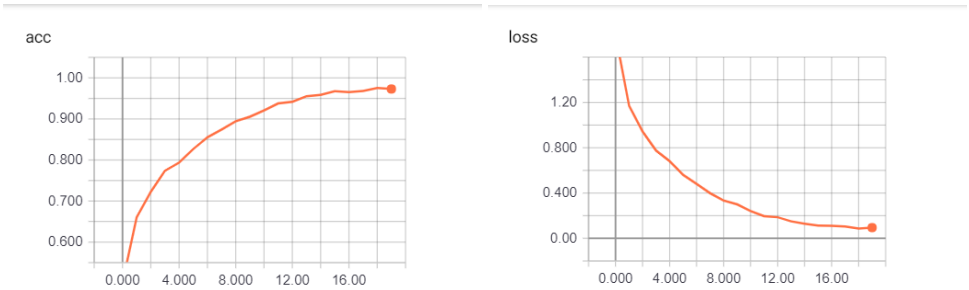
millions of samples and the second only 4774. In these processes, expecially when we use such big models, the amount of data is crucial for the effectiveness of the classifier. Moreover the training process was extremely slow, due to the huge number of parameters to train. The convolutional neural network presented here has a simple structure and relatively few parameters, and in this specific domain it outperforms the two mentioned models.

In figure **??** there's on the horizontal axis the epoch number, and on the vertical axis the metric used to measure performance. In figure 2a one metrics are evaluate on the training set, and we get a very high accuracy and small error. More important are the statistics in figure 2b, where metrics are evaluated on the test set. We've still good results but from the shapes of the two curves we can observe that the algorithms learns a lot until the seventh epoch and then starts to overfit, in fact after that point performances evaluated on training set continue to increase while on the testset they get worse, in particular we have that accuracy fluctuates around the same value and the error increases.

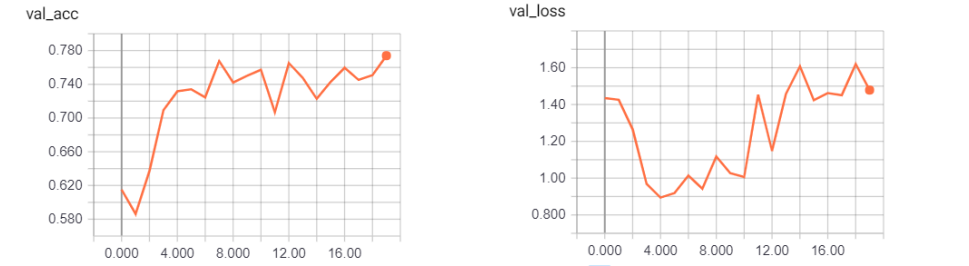
4 Pre-trained CNN

4.1 Model Structure

In section 3 we observed that training big models like InceptionV3 on our dataset does not bring good results. In this section we'll describe a different usage of those models. We'll use InceptionV3 for its excellent results in image classification, and we'll download the model already trained on ImageNet. This can be easily done with Keras [4], when we create the model there's



(a) trainingset



(b) testset

Figure 2: accuracy and loss

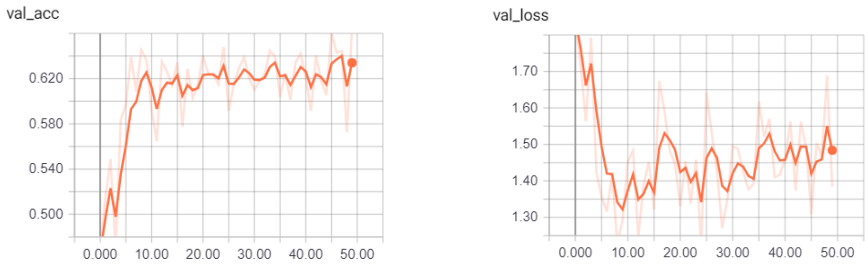
a flag that allows to specify if it must be initialized with weights obtained by training that model on the specified dataset or at random, to train the network from scratch. Imaget[5] has a hierarchical structure, each category can have more specific subcategories, but they are not specific enough to be used directly for our classification problem. In fact the pretrained network as it is can diversify sea boats from river or canal boats or a torpedo-boat from a cruiser but there's no chance it can correctly classify all different categories of Venice boats. We can use instead this net to extract relevant features from an image. The output of each layer can be interpreted as a different representation of the original image. The output of the input layer indeed is the image itself, a matrix of pixels, and the output of the last layer is a label that provides a conceptual representation of the image. If we analyze the output of each level starting from the input and going deeper in the network also the representation moves from a physical level to a more conceptual level. Now the challenge is to choose the output of the intermediate level that provides a representation that has the right "distance" both from the input and the output. In general in convolutional neural network we have first convolutional interleaved with pooling layers, and then one or more that are fully connected. The first two kind of layers have the purpose of feature extraction and dimensionality reduction respectively, while the third one has the purpose of classification. In our model we'll train only the last dense layer in order to take advantage of the ability of extracting features acquired by InceptionV3 while training on a much bigger dataset.

4.2 Performances

This model have been trained for 50 epochs. Comparing the graphs in figure 3a with those in figure 3b is evident that just after 10 epochs the model starts to overfit, because after that point we have that accuracy and loss improve only when measured on the training set.

The relatively big number of epoch will not negatively affect the performance of our model since we saved the model overwriting the old one only in epochs in which we had an increase of the performance measured on the testset. Hovever these results suggest that this model is not the optimal one, since it's too prone to overfitting. Starting from this model we can try to make some modifications in order to observe the corresponding variations on the metrics. For example we can vary the number of neurons in the last dense layer, or we can try to train also some of the last convolutional layers in addiction to the last dense layer. We can also try different optimizers with different parameters or add more than one dense layer at the top. Another

(a) trainingset



(b) testset

Figure 3: accuracy and loss

measure to reduce overfitting is dropout, that consists in making random cuts, different at each iteration, to the input of the dense layer. However experiments shows that it's very difficult with this model to get an accuracy higher than 70%

5 CNN with SVM

5.1 Model Structure

This model, in analogy with the previous one, uses a pre-trained convolutional neural network to extract features, but this time the classification task relies on a SVM. Support Vector Machine is an efficient linear model, but that means that it's able to classify instances under the assumption that they are linearly separable. In other words SVM will work only if there exists an hyperplane that's able to separate instances that belongs to different classes. Actually it's possible to go beyond this assumption using a kernelized version of SVM, that consists in applying a non-linear function on the input. This method uses the fact that in SVM formulation we've that the input appears always in form of product $x^T x^{'}$, so if we sobstitute every occurrence of $x^T x^{'}$ with the term $k(x, x^{'})$ where k is a nonlinear function $k : (x, x^{'}) \to \mathbb{R}$, we obtain a model that it's still linear but not respect to the input. However this approach generates the problem of finding the right function k such that the linear separability hypothesis holds, but the idea behind this model is to leave the hard work of linearizing the input to the convolutional neural network, and then using an SVM without kernels to learn to classify the output of the network. We'll use again InceptionV3 without the last dense layer to extract features, and then LinearSVC [6] (the sklearn implementation of the SVM with linear kernel) to classify.

5.2 Performance

This method outperforms all others in term of accuracy. This result validate the hypothesis that convolutional neural network accomplished quite well the task of linearizing the input. Another point of favor is that it does not require training the convolutional neural network, but only the SVM, that's much more fast.

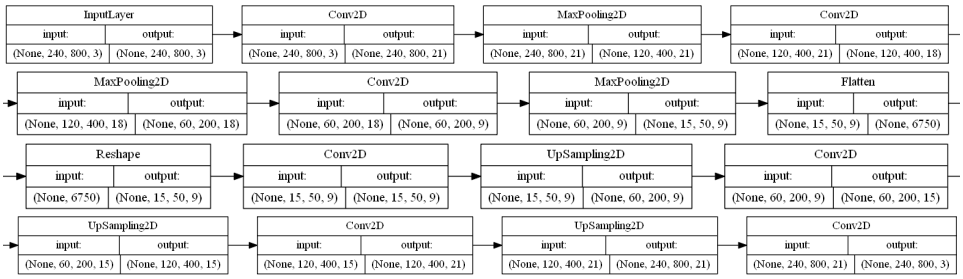


Figure 4: Autoencoder structure

6 Autoencoder

6.1 Model Structure

In this section we'll analyze a completely different strategy. First we'll use an autoencoder to reduce the dimensionality of the input, and then a simple artificial neural network to classify the encoded instances. Encoders have the following structure:

$$Input \rightarrow \boxed{\text{Encoder}} \rightarrow encoding \rightarrow \boxed{\text{Decoder}} \rightarrow Output$$

. The hallmark of an autoencoder is that encoding and decoding functions are not fixed, but parametric, and our model will learn those parameters training on the dataset. More formally, we're going to implement a neural network that will learn the identity function $f : x \in D \rightarrow x$ for the instances in our dataset. We'll do that with a convolutional autoencoder, whose structure is represented in figure 4. Since the output of the encoder must be as similar as possible to the input, input and output layers have the same shape. Then we have a sequence of convolutional layers interleaved with pooling layers, this block will implement the encoding function and its output will be the encoded image. Convolutional layers introduce the parameters of our encoding function, while pooling layers reduce the dimension of the image. Then we have another sequence of convolutional layers interleaved with upsampling layers. This other block is the decoding function, where convolutional layers still have the purpose to introduce parameters and the upsampling layers do the opposite of pooling layer, applying a fixed operation (in our case bilinear interpolation) to the input to augment both height and width of each channel of a factor that's equal to a given size. Considering that we have a input image with $shape = (240, 800, 3)$

Figure 5: Decoded images

while the output of our encoder is a 6750-dimensional vector, we have a $compressionratio = \frac{240 \times 800 \times 3}{6750} = 85 : 1$. The next step is to train an artificial neural network to classify the encoded images. We'll use an ANN with 3 fully connected layers (excluding input and output layers) with respectively 1000, 1000 and 500 units.

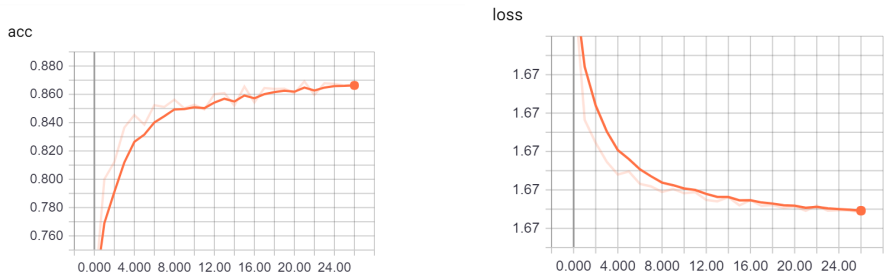
In figure 5 we have on the left the original images and on the right the output of the decoder, that's a reconstruction of the originals from the encoding.

6.2 Performances

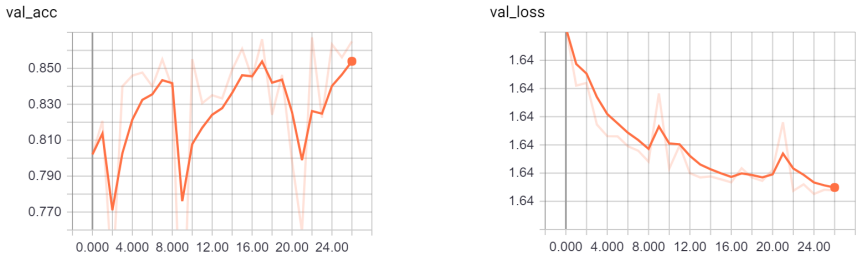
In figure 6 and 7 are reported accuracy and loss during training of both the autoencoder and the artificial neural network used as classifier. We can observe that after 20 epochs of training we get the best accuracy from the classifier. The accuracy of the autoencoder instead oscillates a lot, however this is not so important since we're not interested in the precision of the decoded image but rather to have a minimal representation of the input.

7 Conclusion

In figure 8 there're the confusion matrices of each model. On the vertical axis we have the true values and on the horizontal axis the predicted ones. The element $M[i, j]$ counts the number of instances of class i predicted as j. Each value of the frequence of the couple i, j is assigned to a color according

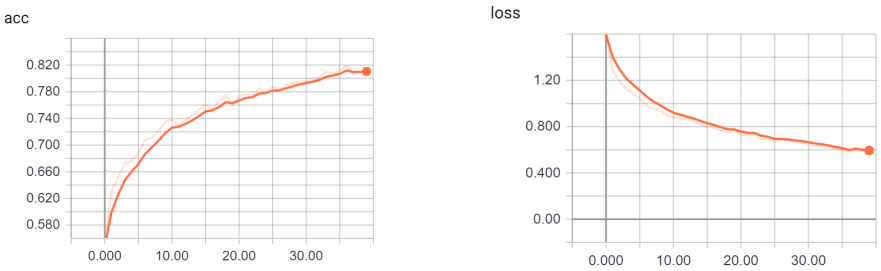


(a) trainingset



(b) testset

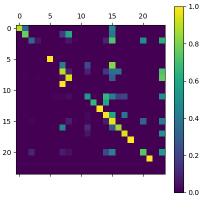
Figure 6: autoencoder accuracy and loss



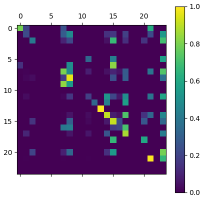
(a) trainingset

(b) testset

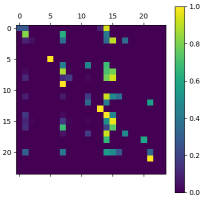
Figure 7: ANN accuracy and loss



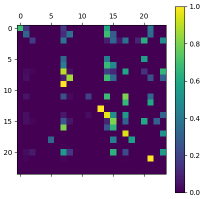
(a) CNN with SVM



(b) CNN



(c) pre-trained CNN



(d) autoencoder

Figure 8: confusion matrices

Classes	Models			
	CNN with SVM	**CNN**	**Pre-trained CNN**	**Autoencoder**
Alilaguna	0.90	0.79	0.26	0.68
Ambulanza	0.77	0.23	0.82	0.32
Barchino	0.33	0.41	0.39	0.18
Cacciapesca	-	-	-	-
Caorlina	-	-	-	-
Gondola	1.00	0.00	1.00	0.00
Lanciafino10m	0.00	0.00	0.00	0.00
Lanciafino10mBianca	0.89	0.79	0.86	0.86
Lanciafino10mMarrone	0.90	0.76	0.14	0.76
Lanciamaggioredi10mBianca	0.00	0.00	0.00	0.00
Lanciamaggioredi10mMarrone	-	-	-	-
Motobarca	0.51	0.15	0.15	0.14
Motopontonerettangolare	0.67	0.33	0.00	0.00
MotoscafoACTV	1.00	1.00	1.00	1.00
Mototopo	0.95	0.81	0.81	0.84
Patanella	0.80	0.50	0.35	0.47
Polizia	0.33	0.07	0.00	0.00
Raccoltarifiuti	0.90	0.53	0.05	0.95
Sandoloaremi	1.00	0.00	0.00	0.33
Sanpierota	-	-	-	-
Topa	0.28	0.00	0.00	0.00
VaporettoACTV	1.00	0.98	0.98	0.94
VigilidelFuoco	-	-	-	-
Global Accuracy	0.85	0.72	0.65	0.73

Table 2: Models Accuracy

to the color map on the right of each matrices. In particular $M[i, j]$ will be yellow if all instances of class i are classified as j, violet if zero instances of i are classified as j or an intermediate color if some instance of i are classified as j. Using the confusion matrices we can compute the accuracy for each class as follows: $accuracy[i] = \frac{M[i,i]}{\sum_j M[i,j]}$. In table **??** are reported those values. When there're no samples of a class in the trainingset the corrisponding row is left blank.

References

[1] D.D.Bloisi, L.Iocchi, A.Pennisi, F.Previtali *MarDCT - Maritime Detection, Classification and Tracking data set* http://www.dis.uniroma1.it/ labrococo/MAR/

[2] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna *Rethinking the Inception Architecture for Computer Vision* https://arxiv.org/abs/1512.00567

[3] Karen Simonyan, Andrew Zisserman *Very Deep Convolutional Networks for Large-Scale Image Recognition* https://arxiv.org/abs/1409.1556

[4] *Keras:Applications* https://keras.io/applications/

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li and Li Fei-Fe. *ImageNet: ALarge-ScaleHierarchicalImageDatabas* http://www.image-net.org/papers/imagenet_cvpr09.pdf Dept. of Computer Science, Princeton University, US

[6] https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html []