Venice Boat Classification

Pietro Spadaccino, 1706250

Homework 2

1 Introduction

Our goal is to implement a classifier for different kinds of boats in Venice. This classifier should be capable of telling whichever class a boat belongs to and must be able to work properly only by observing an image of the boat provided by a webcam, without the use of other external or handcrafted features. We will show how to implement such a classifier and how to train it on the ARGOS dataset.

2 ARGOS dataset

This dataset includes images of around 5500 boats, organized into 23 different classes. The dataset is already split in training and test set, with a 2/3 ratio. The training set is structured in 24 folders, one for each class and another one containing false positives samples of water, which we won't make use of.

Each sample is a 240x800 .jpg image with colors. Since we don't have access to any other information, the whole images will be given to the classifier. The size of the classifier's input will then be 240x800x3 = 576000, considering the size of the image and the RGB channels.

3 Problem defining

We want to implement a classifier that, given an image of a boat, determines correctly the class among all the 23 possibilities. More formally, we have a training set $D = \{(x, y)^n\}$ where x is a vector encoding an image with each pixel normalized in [0,1] and y is a one-hot encoded vector containing the class of x. We want to learn a function $\hat{f} : X \to Y$ where $X = \{x \in \mathbb{R}^{576000} \mid 0 \leq x_i \leq 1\}$ and $Y = \{y \in \mathbb{R}^{23} \mid y_i = 1 \wedge y_{j \neq i} = 0\}$.

4 Problem approach

Observing the dataset, it is clear that the requested classification is not easy. First, many boats seems similar even though they belong to different classes,

secondly, the size of the dataset is not so big, so training a classifier from scratch could be inefficient. For this reason, I decided to use transfer learning, using some pretrained models that solve another similar problem, like object recognition, and then build my classifier on top of that. This can yield better performance and should even be faster to train.

5 Convolutional Neural Networks

In this section we briefly introduce what convolutional neural networks are and how we used them.

Inside a convolutional layer, a kernel (usually more than one) "passes" over the whole input, yielding a new output where each element of the input is modified with respect to its neighbors. When working with images, this kernel can be seen as a fixed-size sliding window that passes over all the pixels producing a new image where each pixel is a function of its neighbors pixels in the input image. This makes the convolutional layers very useful when extracting features from an image, since they consider the ensemble of the pixel in a neighborhood.

Also, convolutional layers need a small number of parameters with respect to fully connected layers: their training just needs to find the parameters of the kernel for each kernel in the layer. As an example, let's consider a layer with 256 kernel filters of size 3x3 compared to a dense layer having input and output size equal to 1000: in the first case we have just to learn 3x3x256 = 2304 parameters, in the latter we would have 10^6.

More over, the output of a convolutional layer with one kernel has often a similar size of the input image, depending on the size of the kernel, the stride (after passing on a region of the image, how many pixels to translate the kernel for applying it on the next region) and the padding added to the image, which is necessary in case the size of the image minus the size of the kernel is not a multiple of the stride. Indeed the less trainable parameters compared to a fully connected layer comes at the expense of the output size: usually we have hundreds of filters in a convolutional layer and the output size will scale linearly with this number. For this reason, usually a pooling operation is added at the end of a layer, reducing the size of the output of each kernel.

5.1 Inception v3

Inception is a very popular neural network designed for object recognition made by Google. It is capable of recognizing object belonging to thousands of different classes with a top error rate of 3.46%, as stated in [4].

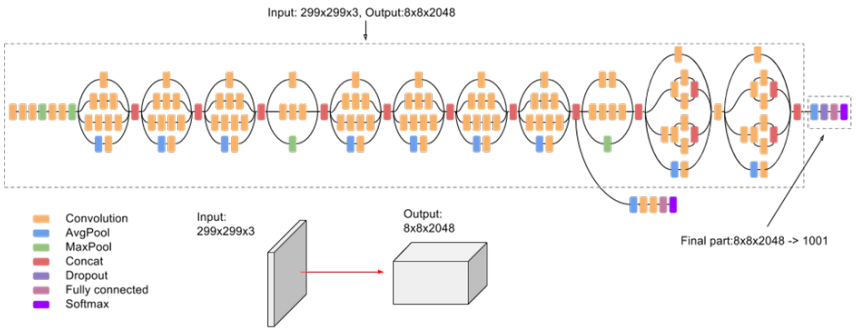


Figure 1: Structure of Inception v3

6 Model defining

My final model will be composed by two different models: the first one, the one to give the image to, must be an already trained one and the second will learn the classification task through transfer learning from the first. I decided the first model, to be a convolutional neural network. By doing so, this net can extract the most relevant features from the image and another model will combine these in order to output a classification. I used an Inception v3 as the first model, with already trained weights. I removed its last layer thus giving me access to the deep features of an image. I can then attach another model, that can be a SVM or another neural network, to be trained on these features and it will yield the final prediction.

When doing a prediction, the input image passes through the whole pipeline: first into the already trained Inception v3, until the second-last layer, and then into the final model which will output the prediction. Instead, the training phase is performed only on the second model, because the first one was already trained. Note that the largest part of the whole model is composed by the Inception network while the "head" model is just a smaller part, for this reason the training should be relatively quick.

7 Implementation

The implementation of the code was made in Tensorflow + Keras for neural networks, sklearn for SVMs and some utility functions provided by tflearn for parsing and loading the dataset.

The implementation follows this pipeline:

1. Parse and load the ARGOS dataset, `dbloader.py`. In this phase we read from disk the provided images and store it in a format that will be later reusable by model training. This parsing is done by the tflearn function `build_hdf5_image_dataset`, which stores the parsed data in an external

`.h5` file. This file ended to be large several gigabytes for the training set only, adding a consistent overhead to the original dataset, which was only half a gigabyte large when uncompressed. But this more space brings more efficient RAM consumption and more performances when we will extract the features in the next phase.

2. Extract deep features from the pretrained net, `extract_features.py`. Keras makes available a model of Inception v3 with pretrained weights on the huge ImageNet dataset [5], and we also have the possibility to directly load the model without the last layers. We are now ready to extract the features of both training and test set: all the images are passed through this network and we collect its output, as described in section 7.1. We now can dump the retrieved features on the disk and we can proceed to train our final model.

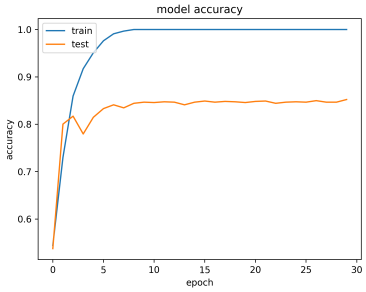
3. Train the final model on extracted features, `svm_final.py` and `net_final.py`. Now that we have the extracted features we can train the final model, which can be a another neural network, in `net_final.py`, or it can be a SVM, in `svm_final.py`. Note that because of how we built the code, both models have in common the first two phases, since they depend only on the features of Inception v3.

7.1 Dataset transformation

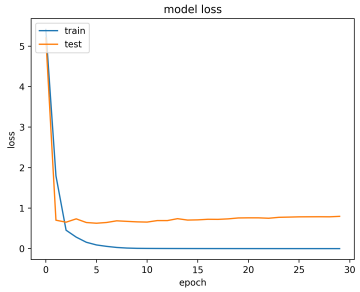
As we said in section 6, the training is performed only on the "head" of the whole model. In order to perform such a training, we have to operate a transformation on the dataset, from $D = \{(x, y)^n\}$ to $F = \{(f, y)^n\}$, where each element x is transformed into its deep features representation f, while y elements remain the same. This transformation is done just by passing x through the layers of Inception v3 except the last one, then attaching a pooling layer to it in order to reduce the output size and finally obtaining a flat vector of size 10240. Now that we have the training set F we can train a model, which can be a SVM or another neural network. By saving the transformed dataset on disk, we can change hyperparameters of the model without having to recompute the features of the original dataset.

8 Results

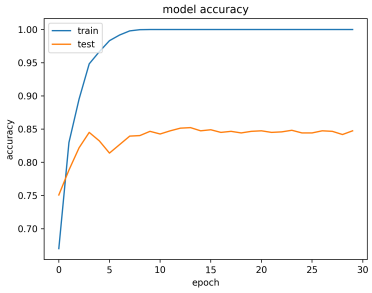
In this section we show the results of our trained models, comparing different tested models.



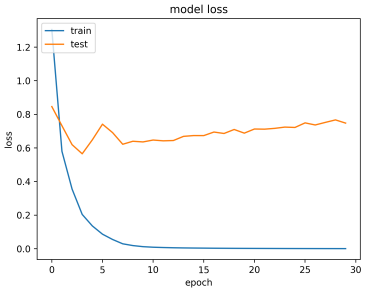
(a) Hidden layer of size 1024, accuracy



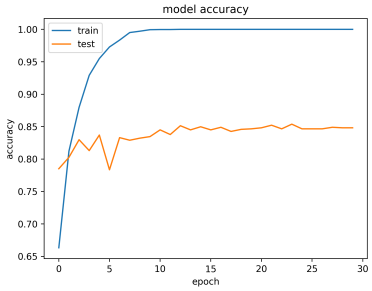
(b) Hidden layer of size 1024, loss



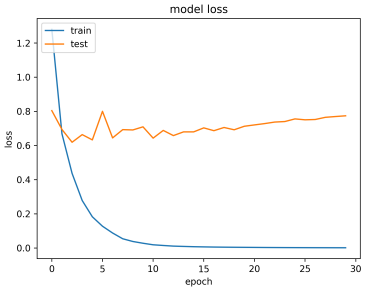
(c) Hidden layer of size 256, accuracy



(d) Hidden layer of size 256, loss



(e) Hidden layer of size 64, accuracy



(f) Hidden layer of size 64, loss

Figure 2: Plots of the accuracy and the loss of different neural networks during 30 epochs of training. All networks are composed by a single hidden layer of variable size and a logits layer. As expected, the more epochs we perform the more the model is prone to overfitting, as seen in the rising values of the loss value in the test set.

8.1 Neural network from deep features

As first model we have trained a neural network based on the features extracted by Inception v3. We trained different kinds of neural networks, varying hyper-parameters like number of layers and number of perceptrons per layer. While the structures of the tested networks are different, they share the same input and output layer. The input layer is a tensor of size equal to the output of the second last layer of Inception v3 after an application of a average pooling layer and then flattened, resulting in a vector of size 10240. The output layer is a logits layer of size 23 (equal to the number of classes). In this layer, softmax activation function is applied, returning the probabilities p_i that the sample belongs to a certain class i.

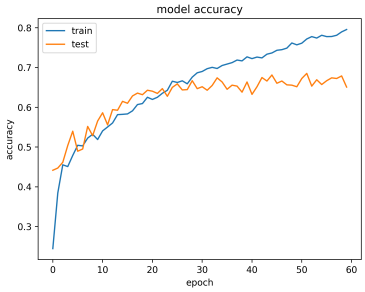
In figure 2 are reported the accuracy and the loss of different neural networks during the training phase. We used networks composed by a single hidden layer and the final logits layer and we can see that there wasn't any substantial difference in performances by varying the size of this hidden layer. We also tried to stack multiple hidden layers instead of a single one, but again, we didn't observe a performance improvement.

8.2 SVM from deep features

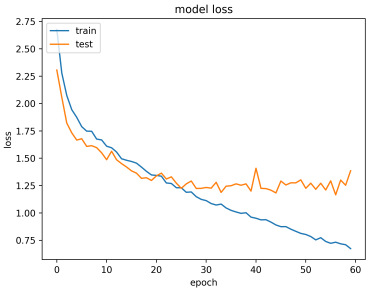
We trained a linear SVM based on the extracted features of the Inception v3. The overall accuracy was good, on the same level as neural networks described before. The detailed accuracy for each class is reported in 1.

8.3 Neural network from scratch

As a baseline, we also tried to train a LeNet instance on the original dataset, without any prior feature extraction. The net was a slightly modified LeNet, with some extra pooling to make it fit inside the VRAM of the GPU and it was trained with a fairly low learning rate. The training phase, reported in figure 3, was more complicated compared to the neural networks trained on deep features. As expected, the performances were not so high and they are reported in table 1.



(a) LeNet, accuracy



(b) LeNet, loss

Figure 3: Plots of the accuracy and the loss during LeNet training, lr=0.0001. The trained obviously took more epochs and more time per epoch when compared to the other networks using pretrained features.

Table 1: Accuracy of different boats classes for different models measured on test set. From left to right: neural network with a hidden layer of 1024 perceptrons trained on deep features, linear SVM trained on deep features, LeNet trained on the original dataset from scratch. The dash "-" means that there wasn't any sample of that boat class in the test set.

	Accuracy on test set		
Classes	NN on features	SVM on features	LeNet from scratch
Alilaguna	94.7%	**100%**	40.7%
Ambulanza	**84.0%**	80.0%	17.6%
Barchino	60.0%	**64.0%**	12.2%
Cacciapesca	0.0%	0.0%	0.0%
Carolina	-	-	-
Gondola	**75.0%**	**75.0%**	0.0%
Lanciafino10m	0.0%	0.0%	0.0%
Lanciafino10mBianca	87.8%	**88.1%**	77.5%
Lanciafino10mMarrone	**86.6%**	84.6%	65.7%
Lanciamaggioredi10mBianca	-	-	-
Lanciamaggioredi10mMarrone	-	-	-
Motobarca	**63.8%**	**63.8%**	22.2%
Motopontonerettangolare	**100%**	**100%**	40.0%
MotoscafoACTV	**100%**	**100%**	0.0%
Mototopo	**89.3%**	87.9%	58.0%
Patanella	51.9%	**52.3%**	32.7%
Polizia	**54.5%**	**54.5%**	0.0%
Raccoltarifiuti	73.6%	**76.1%**	10.0%
Sandaloaremi	**100%**	**100%**	0.0%
Sanpierota	-	-	-
Topa	**64.7%**	50.0%	17.6%
VaporettoACTV	99.6%	**100%**	90.0%
VigilidelFuoco	0.0%	0.0%	0.0%

9 Conclusions

We have implemented and compared different models for classifying different kinds of boats in Venice. We have used a neural network and a SVM and we have shown that the accuracy of the final model did not change a lot, given that they both started from the same deep features. In both cases the performances were similar, and far greater than the accuracy of a neural network, in our case LeNet, trained from scratch.

References

[1] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis and al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, *tensorflow.org*

[2] François Chollet and al. Keras, *https://keras.io*

[3] Aymeric Damien and al. TFLearn, *github.com/tflearn/tflearn*

[4] TensorFlow, Inception *tensorflow.org/tutorials/images/image_recognition*

[5] Deng, J. and Dong, W. and Socher, R. and Li, L.-J. and Li, K. and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database, *image-net.org/*