Rock-Paper-Scissors classifier

Introduction

Guglielmo Cassini 492731, Academic Year 20/21

Image recognition in computer vision is one of the most interesting tasks of computer science in the current years and from 2012 to nowadays it has been improved by the use of machine learning classifiers, in particular with the use of deep learning neural networks.
Given an RGB image of size 224x224 pixels, our goal is to classify which is the move done by a human playing the rock-paper-scissors game.
Exploring the dataset we can see how all images contain the hand of the player, with a green background. The images have been taken from above the hand, and more or less the hands are all centered in the image.

Preprocessing

Before applying any machine learning algorithms to the images, we apply some preprocessing to the data. We mention the fact that the dataset has been already subdivided into train, validation and test dataset, therefore it is not necessary to do it by ourselves.
For each image we apply different techniques to get some low level features of the image, in particular we use:

1. **color histogram**: given the image, it computes the histogram of each color channel, to get a single vector of feature for the color image.
2. **edge direction histogram**: given the image, the descriptors describe the local orientation or edge directions.
3. **co-occurrence matrix**: given the image, it translates the image to gray level image and then counts how many times the same pair of pixel values are adjacent in the image, storing these values in a matrix.
4. **RGB co-occurrence matrix**: like the co-occurrence matrix, but it uses the original RGB image instead of transforming into a gray level image before applying the techniques.

These are well known techniques in literature, designed in origin for computer vision algorithms that don't use machine learning techniques, but are commonly used also for these.
To apply each of these techniques we use a python script called *feature_extraction.py* that requires as argument on the command line an integer number from 0 to 4. Using numbers from 0 to 3 we can choose which of the previously described preprocessing techniques to apply to our train, validation and test datasets and to store it in the *dataset_processed* folder.
As a matter of fact another script inside the folder *script_runner* called *feature_extraction_runner.py* has the task to run multiple times the *feature_extraction.py* script with different arguments to compute all the low level features such that we can try each of them when training our classifiers.
A first comment we can do is that, being all images with the same background and expecting that the human color skin should not be so important as the shape of the image, we would expect that better results are obtained from techniques that depend only on the shape assumed by the hand and not on the pixel colors of the image. Anyway we were wrong, but we will see later.

Machine learning models and tuning

For our classification problem we decided to use two different models:

1. Kernel support vector machines
2. Multilayer perceptron neural networks

A SVM is a linear classifier whose goal is to find a hyperplane that maximises the margin between the plane and the two classes, but it is soft so it allows some points to be inside the margin till a certain point, using a penalization term.
The kernel support vector machines are non linear SVMs that use a kernel function to map from a linear space to a non linear space our features but computing all of this still in a linear space, are useful when the data is non linearly separable.
Usually the image classification is not suitable for linear models, because assigning a weight to each pixel leads the model to be not invariant to some transformations like rotations, mirroring ecc…
We use a KSVM, using the radial basis function kernel, therefore we have to tune the gamma parameter, representing how far are the points that will affect the classification of a certain point we want to classify.

A multilayer perceptron neural network is a feed forward neural network, so a neural network where cycles are not allowed. At start we will try the use of some simple NN without hidden layers, therefore the results will be a linear classifier. Anyway we will see that there are results very interesting even using a linear classifier with the low level features we presented.

To train the KSVM we use a script called *train_ksvm.py*, here we can find a KSVM_OVO class that wraps the methods of the pvml library. As a matter of fact, having few classes we decided to use the one versus one strategy for the ksvm, that consists of creating a classifier for each pair of classes we have in our multiclass problem, and the point is classified to the class that is voted more times from the pair classifiers.
The script has 4 arguments: the first is an integer argument telling which of the low level feature dataset to use, the second is the value of lambda to generalize the model, the value of the gamma of the radial basis kernel function and the value of the learning rate.
At the end the model will be stored using the pickle library and its train and validation accuracies will be saved in a json dedicated to all KSVM models called *results/ksvm_results.json*.
Having 3 many parameters to tune and to choose which low level features to use, using the grid search algorithm to find the best combination of hyperparameters will be too time consuming. Therefore we decided to randomly select the combination of hyperparameters to try. For each low level feature we try 10 random combinations of hyperparameters.
To perform this random search, we use the *script_runners/ksvm_runner.py* that applies exactly this reasoning, running more times the *train_ksvm.py* with different arguments.

To train the MLP NN we use a script called *train_mlp.py*. It requires which low level feature to use, for how many epochs we must train the network, the size of the batch since we use the mini-batch stochastic gradient descent and the value of the learning rate.
To decide hyperparameters to use to train the network, despite being again 3, the values assumed by the epochs we decided to use are only 100, 1000 and 10’000, while the values assumed by the batch-size are 5 and the values assumed by the learning rate are 9.

Therefore we can afford to use grid search to try different combinations of hyperparameters. To do this use the *script_runners/mlp_runner.py* script. Again the results of each model in terms of train and validation accuracy will be stored in a json called *results/mlp_results.json,* while each model will be stored under the *model* folder.
As a matter of fact, we decided for this model to use the default values of lambda and of the momentum. To improve this experiment could be interesting to also tune it.

Results

At this point we have trained different models with many different parameters.
For each low level feature function applied to our dataset, we took the model with best validation accuracy and put it in the following table.

	model	kfun	lambda	kparam	lr	function	train	validation
0	ksvm	rbf	0.001	0.01	0,1	color histogram	0,3442	0,3333
6	ksvm	rbf	0.001	0.2	0,1	co occurrence matrix	0,3241	0,3333
3	ksvm	rbf	0.002	0.2	0,1	edge direction histogram	0,3241	0,3333
9	ksvm	rbf	0.002	0.03	0,1	rgb co occurrence matrix	0,3241	0,3333

As a matter of fact, from our results we don't get a model that wins over the others, are all around the same train and validation accuracy. Also we mention the fact that we thought that the color wasn't important for the classification, since we expected the fact that the shape of the hand was more relevant, but as a matter of fact the color histogram got very close performance to the models that use features without color information.
Another thing we can say is that all these models are not overfitting, since the train and validation accuracies are very close, but we have no idea at this point if models are underfitting because we don't know the optimal bayes classifier performance, but looking to successive results got with MLP we can asses that they're underfitting.
Note that the fact that for second, third and fourth rows, the fact that validation is higher than train is just an error due to computational approximations.

As before, for each low level function we took the MLP model that maximises the validation accuracy in that group. The results are summarized in the following table.
As we can see with the use of multilayer perceptrons we get much better results compared to the ones of the KSVMs.
As a matter of fact, in this case again we're not overfitting since train and validation are close, even if there are some random differences due to the use of the stochastic gradient descent for the training of the network, and this also explains why sometimes the validation accuracy is greater than the train one. Anyway we still have no idea if we're underfitting or not.
We can see how again our prejudice against the "colored features" were wrong, since the model with highest accuracy in training is the one trained with the rgb co occurrence matrix features (but anyway the color histogram is the one with with lowest accuracy...).

	model	feature func	layers	lr	epochs	batch size	train	validation
25	mlp	color histogram	[192, 3]	0.2	100	30	0,7256	0,7333
226	mlp	co occurrence matrix	[64, 3]	0.002	10000	10	0,8241	0,8533
211	mlp	edge direction histogram	[64, 3]	0.02	10000	10	0,7981	0,8333
238	mlp	rgb co occurrence matrix	[729, 3]	0.02	1000	10	0,8755	0,9066

With these we conclude the comments about low level features. To obtain these tables it is necessary to run the *best_ksvm_and_mlp_for_each_low_level_feature.py* scripts that will save these two tables inside 2 excel files.

CNN Feature extraction

With the rgb co occurrence matrix we got interesting results, but we can decide to try another method to see if we can do better.
We use a technique called feature extraction using a convolutional neural network: it consists in taking a pretrained CNN, cutting off the last layer, to infer the image and take the output of the network. This will be a new feature vector describing the input image and that we can use as input to another classifier.

The CNN network we use will be the one provided during the cake classification laboratory, composed of a sequence of eight “same size” convolutions followed by three fully-connected layers. All layers are followed by ReLU activations. Some convolutions (S) are strided. A final softmax computes the class probabilities. The network has been trained on the ILSVRC-12 subset of ImageNet.

To obtain this features that we call “deepfeatures” inside our code, we need to run the *feature_extraction.py* script passing the value 4 as first argument. This is done by the *script_runners/feature_extraction_runner.py* script.

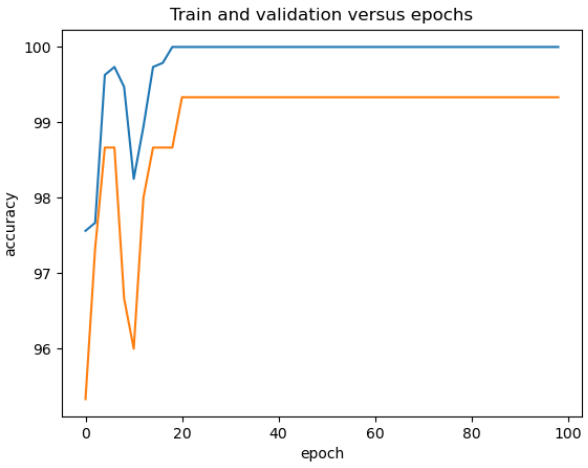
We use these new features as input for both the KSVM and the MLP and the results are summarized in the following table, showing the best results among all models.
The use of these features is done using the *script_runners/ksvm_runner.py* and *script_runners/mlp_runner.py* scripts, that repeat the same procedure described for the low level features.
At the end we can summarize the results in the following tables.

	model	feature func	layers	lr	epochs	batch size	train	validation
255	mlp	deep features	[1024, 3]	0.01	100	10	1	0,9933

	model	kfun	lambda	kparam	lr	function	train	validation
14	ksvm	rbf	0.03	0.03	0,1	deep features	0,9666	0,9242

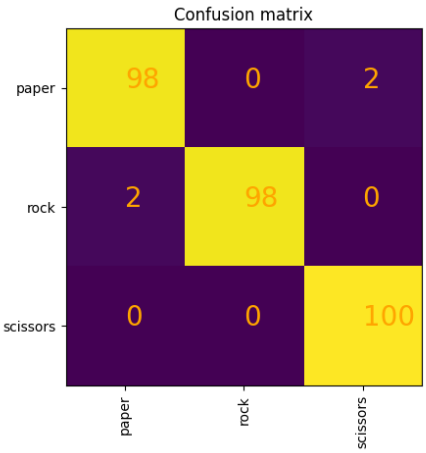
The results obtained using this kind of feature are really satisfactory, with both being above the 90 percent of accuracy. In particular the MLP reaches a validation accuracy of 99,33%. None of the two models is overtiffing, being very close in train and accuracy, and we assume that the MLP model is neither underfitting (or at least not so much) having an accuracy practically close to 100%, so maybe we can assume that we're very near to the abilities of the optimal classifier for this problem.

Accuracy curve, confusion matrix and conclusions



After all our experiments, the best model is a MLP with only 100 epochs of training. But are 100 epochs really necessary? Our objective is to give a look at how the accuracy curve changes with respect to the number of epochs used to train the classifier.
Looking at the plot on the left we can see that the validation accuracy doesn't improve after the epoch 20, therefore we can even stop at that step and reduce the time necessary to train the model.
This plot is built automatically for a specific model during the *train_mlp.py* execution.

Now we evaluate the best model with the test dataset. At first we say that the test accuracy is 98.96%, and then we can give a look to the confusion matrix. We can see that all the samples belonging to the scissors are well classified, while only 2% of rocks are misclassified as paper, while only 2% of papers are classified as scissors.
All the others are correctly classified.
This is created by the *confusion_matrix.py* script.



With this we conclude the experiments of this problem, classify the rock scissors paper move of a player is possible from the image with satisfactory results and would be very interesting to play it in this way.

I affirm that this report is the result of my own work and that I did not share any part of it with anyone else except the teacher.