Programming assignment - Dice - Luca Ferrari

July 14, 2020

1 Feature extraction

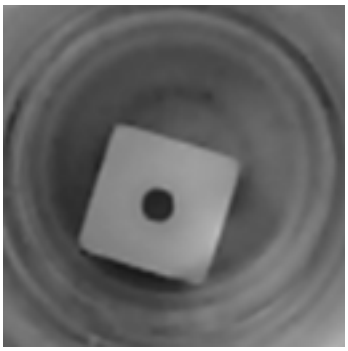
(features_extraction.py for reference)

The objective of the system we need to design is to determine the outcome of the roll by looking at the image of the single die. To train a classification model to complete the task, we need to extract features from the images provided in the data set. By looking at the sample images we can immediately see that:

- dice can be both black and white
- the dots on the faces can be rotated in different ways
- dice can be in different positions in the image(center, left, right..)

By taking into account all these factors, we need to choose features that are invariant to translations and rotations. According to these requirements, the features chosen to be computed are the **gray-level co-occurence matrix** (invariant to translations and rotations), **color histogram** (invariant to the majority of geometric transformation) and the combination of the two. During the developement, other kind of features were first taken into account but then discarded because of technical problems (**CHT - circle Hough transform**) or high computational cost (**neural features**).

*From previous studies I know that **CHT** is a feature extraction technique used to detect circles in images so I tried to use it to detect the dots in die faces, however the images were too small and the smallest circles that could be detected were the ones of the die container in the background.*



*After correctly padding the images to make them usable in **pvmlnet** (128x128 –> 224x224), I tried to extract neural features by extracting the activations after a forward pass in the CNN, but a high computational cost was required to complete the task for all the datasets.(neural_features_extraction.py included but unused).*

2 Modelling

The problem we are facing is an image classification problem. There are different models we can use to complete the task, however we most certainly know that a linear classifier won't be enough to do that (*nevertheless a linear classifier among the other models has been implemented for demonstration purposes*). A CNN could have been used, but since a good accuracy was reached with the following models and since the computational cost of using that would have been really high, I decided not to use it. Models implemented:

- **Multinomial logistic regression** (linear classfier)
- **MLP** (with different architectures)
- **KSVM**

2.1 MLP / Multinomial logistic regression

(classifier_training.py, models_validation.py for reference)

To set a baseline for the accuracy of the classifier I decided to implement a Multinomial logistic regression model (linear classifier) in form of a MLP with no hidden layers *(classifier_training.py)* so that the implementation of an MLP with a more complex architecture would have been faster.

In the best case scenario (best training hyperparameters), for the multilayer logistic regression I got 51.98% in training accuracy and 51.33% in test accuracy which is clearly unsufficient to deploy the system. Some non-linearity needs to be included in the model. Both the mlr classifier and the MLP were trained with the features extracted previously as co-occurence matrix from the training images provided.

To find the best architecture for the MLP and the most performing set of hyperparameters, I designed a validation schema (first step grid search) and wrote a script that tries different set of parameters and architectures on the validation set *(models_validation.py)*. The script saves the differently trained networks named according to the architecture and hyperparameters used in a specific folder and produces a report that show us the training accuracy and the validation accuracy.

(First step)

MLP architectures (neurons per layer):

- [64, 6] - (multinomial logistic regression)
- [64, 32, 6]
- [64, 32, 16, 6]

Hyperparameters:

- **epochs**: 10, 100, 500, 1000
- **batch size**: 80, 120
- **learning rate**: 0.01, 0.001, 0.0001

I let the script run for a big while and then I examined the results. Looking at the reports I could assess that the best performing architecture (as expected because has more hidden layers) was the [64, 32, 16, 6] neurons architecture trained for 1000 iterations, 120 samples batches and a 0.01 learning rate that had a 93.44% training accuracy and a 93.33% test accuracy.

(Second step)

MLP architectures (neurons per layer):

- [64, 32, 6]
- [64, 32, 16, 6]

Hyperparameters:

- **epochs**: 1000, 5000, 10000
- **batch size**: 80, 120
- **learning rate**: 0.01, 0.001

To further exploit the information contained in the data, I retrained the model with a different validation schema (second step) increasing a lot the overall model performance (**training accuracy**: 99.90; **test accuracy**: 99.58). I also trained the MLP with the **color histogram** feature and with a combination of that and **co-occurence matrix**. In the first case the model performance was similar to the one trained with the co-occurence matrix feature. In the second case a fewer iterations were needed to get to a good model test accuracy (~ 96.67) but some overfitting was introduced that could potentially take away space for improvement (**training accuracy**: 99.27 ;**test accuracy**: 96.67). Since the model was already performing pretty well and since I was just using one single feature to determine the outcome of the roll, I decided to implement a simpler model (KSVM) to see how much the training with different features would affect the overall performance of that.

(models and reports can be found in the directory models/a-64-6, models/a-64-32-6 etc.)

2.2 KSVM

(ksvm_training.py for reference)

As already mentioned, I decided to train a simpler model to see how the feature choice would affect the model performance. The model I decided to train was a KSVM because inserting a non-linearity (as we saw before) is crucial to get good results in an image classifier.

To train the model I initially used the **co-occurence matrix** as done with the MLP training, and then I combined that with the color histogram feature extracted before. Since we were dealing with a multi class classification problem I used the OneVsRestClassifier from the **scikit-learn** library in conjunction with a polynomial kernel (also provided in the library) to transform the data.

To choose the polynomial degree to use with the kernel, I made a similar validation script to the one presented before that trains KSVM classifiers with different kernel polynomial degrees and produces a report with training and test accuracy for every single trained model.

The model's accuracy grew as expected by increasing the polynomial degree of the kernel until it reaches a plateau ($\sim 81.25\%$ with $\mathbf{deg} = 16$). Then I tried to re-train the model with a combination of the two extracted features (**co-occurence matrix**, **color histogram**) and the results were very different: with a much lower polynomial degree kernel ($\mathbf{deg} = 6$) I got a higher test accuracy than the one which was the maximum before ($\sim 82.92\%$) and also a much higher overall best (test) accuracy value (~ 92.5 $\mathbf{deg} = 18$).

This confirms the fact that if we decide to use a simpler model, a more accurate feature engineering greatly pays off in terms of model's accuracy.

(reports can be found in the directory models/KSVM etc.)

3 Analysis

(model_analysis.py for reference)

As explained in the report, models were selected according to their accuracy, so a performance analysis has been conducted basically at every step on the validation set. However, to further confirm the results, a model analysis has been conducted also on the actual test set and it was possible to assess that the best model (MLP, a-64-32-16-6-e10000-b120-lr0.001) was able to correctly classify images with a **0.9875 (test) accuracy**. In the *model_analysis.py* script I also found the worst errors *(The script prints 10 of them but only 3 are actual errors because it prints out the first ten predictions with the lowest confidence)*, the confusion matrix (which isn't actually very significant) and the hardest class to identify (class 1, *two dots on die*).

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.