Food Recognition with Traditional Classifiers and Deep Neural Networks

Daniele Montagnani
Universita degli Studi di Milano-Bicocca
Id: 896621
Email: d.montagnani@campus.unimib.it

***Abstract*—In this project I performed an image classification task on the iFood-251 dataset with different methods. First, I considered traditional feature extraction coupled with traditional classifiers. Then, I've followed a deep learning approach by using convolutional neural networks. In particular, I used a simple convolutional neural network and a residual network. I obtained somewhat satisfactory performances with the deep learning approach. With the traditional approaches, on the other hand, I obtained really unsatisfactory performances. However, traditional models' performances were significantly above the random guessing benchmark indicating that some learning, albeit limited, was taking place.**

1. Introduction

The objective of this project is to study the image classification performances of different methods. In particular, the traditional feature extraction + traditional classifier pipeline is compared with the more modern deep learning approach.

Image classification is a subfield of computer vision concerned with developing automatic method to classify images into distinct classes. Its functionalities can be useful for a moltitude of different scenarios. Just to cite a few: medical diagnosys, autonomous driving and automatic industrial quality control.

Automatic methods for image classification need to extract useful features from images in order to proceed with classification. Traditionally, feature extraction has been an automatic but rigid process following human designed rules. With deep learning, however, things have changed. With deep learning deep neural networks are trained end to end (image to label) and automatically learn to extract the most useful visual features for the task at end.

2. Dataset and Data Pre-processing

2.1. The Dataset

For the following study, I used the 2019 iFood Competition dataset. It contains images from 251 fine-grained food categories. The image size is not standardised and varies between images. Every class contains between 100 and 600 items.

As reported on the challenge's description, classification is particularly challenging on this dataset. In fact, there exists high visual similarity between different classes and foods from the same class are presented in highly diverse conditions.

Since the test set was not provided, I had to rearrange the dataset splits. The original validation set, of 11,994 images, was used as the test set for the task. The original training set was split to obtain a validation set (20%, 23695 observations) and a smaller training set (80%, 94780 observations).

In order to manage my dataset, I used the Dataset class from Pytorch to wrap training, validation and test images. Then, for organisational and efficiency purposes, I organised my dataset into a Pytorch Lightning Datamodule. All in all, the Datamodule is a big object that: wraps download and setup functionalities, holds Pytorch Dataset attributes, generates Dataloaders, and facilitates the organisation of custom data-related functionalities. For example, in the Datamodule I wrote custom methods to: visualise some images, extract features and generate datasets to manage features in training.

2.2. Data Pre-processing

Image files are traditionally stored in formats most suited for human fruition through digital devices. Naturally, this formats might not be perfectly suited for machine learning tasks. For this reason some preprocessing steps need to be performed in order to make the data redy to be fed into our models.

In my approach I used different preprocessing pipelines in the traditional and in the deep learning approaches. This distinction is due to different format needs (such as gray vs color) but also to feasibility and timing considerations.

For the deep learning approach I followed a pretty standard preprocessing pipeline. First, images were resized to a standard shape (256 by 256). Then, they were converted to torch tensors. Lastly, they were normalised with the statistics (mean and standard deviation) from the Imagenet Dataset.

For the traditional approach, I followed the same procedure except that images were resized to a 64 by 64 size. Furthermore, before being fed into the SIFT algorithm, images were converted to grayscale.

As we ca see in figure 1, I concluded by visualising some images as a sanity check.

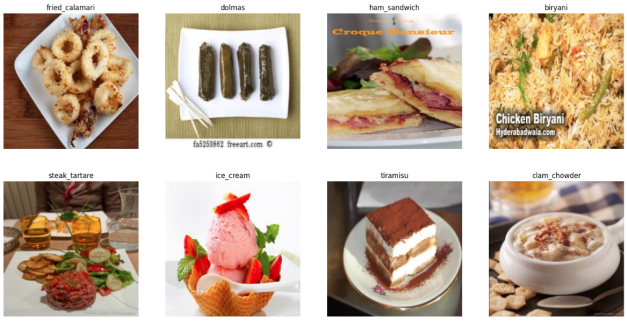

Figure 1. Sample Images

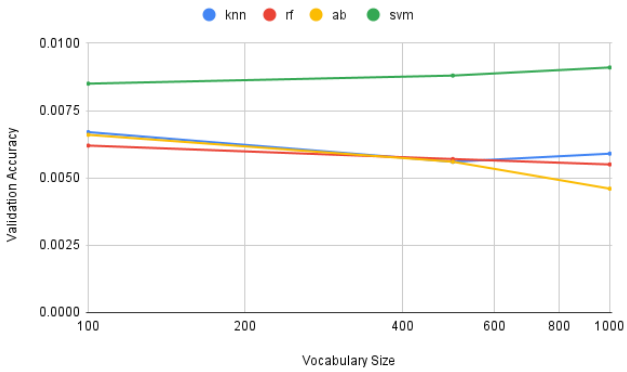

Figure 2. Best Model (as per the hyperparameter search) Performance vs Vocabulary Size

3. Traditional Feature Extraction

To train my traditional classifiers, I used features extracted with traditional algorithmic methods. In particular, I considered the Scale Invariant Feature Transform followed by a Bag of Words Representation.

The Scale Invariant Feature Transform is an algorithmic approach able to detect keypoints and compute descriptors at different scales. This is achieved by considering a Difference of Gaussian transformation at different values of sigma. Effectively, this procedure act as a "feature bandpass filter" to detect features of a particular "characteristic resolution".

The Bag of Words representation describes an image by the frequency of peculiar visual features within it. To follow the BoW approach, one starts from the SIFT transform descriptors. Then, uses a clustering algorithm (K-means) in order to find "prototypical features", also known as "the vocabulary", in the algorithm centroids. Lastly, images descriptors are clustered to form an histogram representation. Ultimately an image is described by the frequency of features in each vocabulary class.

To implement execute the SIFT procedure I used the functionalities already provided in the open-cv library. The BoW procedure, instead, was implemented with custom code. As an external resource I used the K-Means implementation found in sklearn. For computational reasons(RAM), I adoped a mini-batch version able to fit data in batches.

A significant hyperparameter used while computing the BoW representation was the vocabulary size (the number of K-means clusters). to explore possibilities while maintaining computational feasibility I considered vocabulary sizes of 100, 500, and 1000. As we shall see in the next section, however, the vocabulary size was mostly ininfluential towards model performances.

4. Traditional Classifiers

Once I had the BoW representation for each image, I went on to fit traditional classifiers. For the project I considered: Naive Bayes, Support Vector Machines, K-Nearest Neighbors, Random Forest, and AdaBoost.

For every classifier I consulted the documentation in order to understand what hyperparameters were interesting. Then, I timed every model's training process in order to understand what hyperparameter space size I could explore. Lastly I performed an exhaustive search in the identified space in order to find the best hyperarameter combination. For computation reasons, the hyperparameter search was performed on slightly more than half the training images.

The hyperparameter search has been performed considering all vocabulary sizes (100, 500, 1000). But, as it can be seen in figure 2 , where I plot the best models' performance for each class and each vocabulary size, it did not impact performances significantly. For this reason, in order to simplify my exposition, I will expose results obtained by considering a vocabulary size of 100.

4.1. Naive Bayes

As for the Naive Bayes Classifier, I used the Gaussian Naive Bayes implementation. As I did not identify particularly interesting hyperparameters, I did not perform an hyperparameter search and took the performance of the default configuration.

4.2. Support Vector Machines

For the SVM machine algorithm, I considered a "radial basis function" kernel, a "scale" gamma, and a "balanced" class weight. I considered values of the hyperparameter C, in the list: 0.01, 0.1, 1, 10, and 100. The best performance of 86 pips was obtained from the model with C set to 1. In figure 3 we see a depiction of the hyperparameter search with a parallel coordinates chart. A parallel coordinate chart, which I will use extensively in my analysis, is a convenient way to summarise the relationship between hyperparameter configurations and model's performance. Namely, a set of parallel coordinates encode the hyperparameter space over which the search was performed. Then, a curve represents a hyperparameter configuration by passing through all the configuration's hyperparameter values. Lastly, in the last parallel coordinate the performance of the specific configuration is indicated. Conveniently the configuration's performance is also coded into the curves colours.

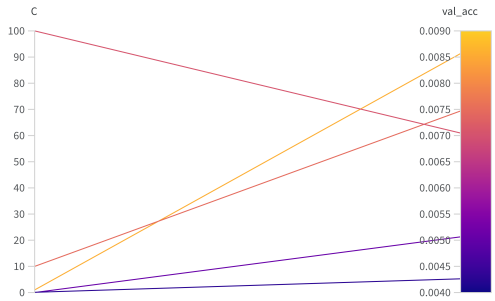

Figure 3. Support Vector Machine parallel coordinates chart

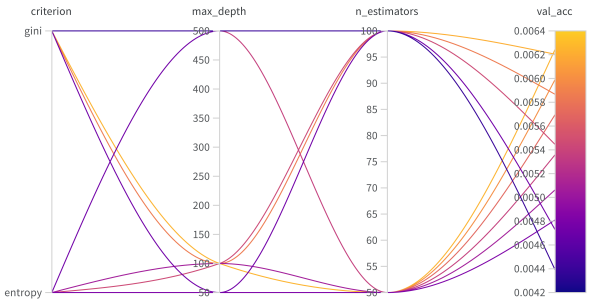

Figure 5. Random Forest parallel coordinates chart

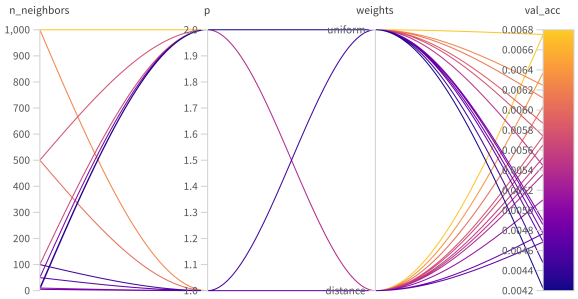

Figure 4. KNN parallel coordinates chart

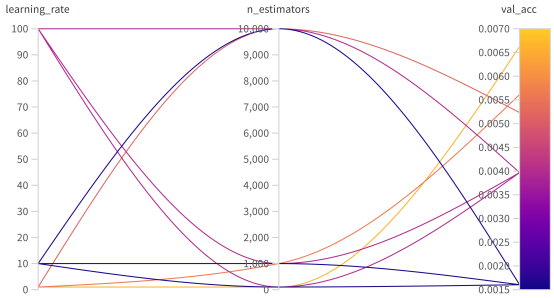

Figure 6. Adaboost parallel coordinates chart

4.3. K-Nearest Neighbors

For the KNN machine algorithm, I considered 5, 10, 50, 100, 500, and 1000 neighbours. Then, Minkowski metrics with p equal to 1 and 2. Lastly, ”uniform” and ”distance” methods to weight neighbours’ importance. The best performance of 68 pips was obtained by considering: 1000 neighbours, p equal to 2, and uniform weighting. In 4 we can see the parallel coordinates chart.

4.4. Random Forest

For the Random Forest Algorithm, I considered two split criteria: the Gini index and the Shannon information gain. I considered number of trees equal to 50 and 100 (I detected a RAM bottleneck). Lastly I allowed trees max depths of: 50, 100, and 500. The best performance (62 pips) was obtained with a ”gini” split criterion, 50 trees of max depth equal to 100. In figure 5 we can see the parallel coordinates chart.

4.5. Adaboost

For the Adaboost algorithm, I considered numbers of estimators in: 100, 1000, and 10000. I also considered learning rates in: 1, 10, 100. The best performance (66 pips) was obtained with a learning rate of 1 and a number of estimators equal to 100. In figure 6 we can see the parallel coordinates chart.

4.6. Final Evaluation

Finally, once I identified the best performing configurations, I trained them on the full training dataset and evaluated them on the test set. In table 7 we can see the final performance of our algorithms.

Notably performances improved with respect to the validation accuracies obtained during the hyperparameter sweeps. This had to be expected as final evaluation was performed on a model trained on almost twice the data.

However, all in all, the performances of traditional classifiers, while significantly above the random guessing

	Accuracy	Precision	Recall	F1-Score
Naïve Bayes	0.0097	0.0068	0.0097	0.0061
SVM	0.0099	0.0069	0.0099	0.0072
KNN	0.0043	0.0042	0.0043	0.0021
Random Forest	0.0051	0.0044	0.0051	0.0042
Adaboost	0.0091	0.0106	0.0091	0.0061

Figure 7. Evaluation Metrics Of Traditional Classifiers

benchmark, are not satisfactory. K-Nearest Neighbors is the algorithm with the worst performance. In retrospect, this had to be expected as it relies on distance while our dataset has a lot of dimensions (an instance of the curse of dimensionality). Clearly, the dataset was too complex to be classified with traditional classifiers trained on images of such low resolution.

5. Convolutional Neural Networks and Supervised Learning

Following the traditional Deep Learning Approach, I trained a Convolutional Neural Network on the classification task.

The rationale behind this approach is to use as a very complex model with an enormous number of trainable parameters in order to achieve better performance. Through the model, the information contained in the data is processed and features are automatically extracted to be used by a final set of classification layers.

A first approach to image classification with deep learning could consider the use of multilayer perceptrons which however, in this case, are extremely inefficient. In fact, MLPs would use an enormous number of parameter and would need to learn the geometry of the image by themselves.

Convolutional Neural Networks greatly increase efficiency, as well as performance, by leveraging the geometrical structure of images and by employing a sort of "parameter sharing". In fact, in convolutional neural networks every layer applies a set of small filters (called kernels) uniformly throughout the image. This way information from pixels is integrated locally. Furthermore, the weights of a single kernel are used to process all pixels in an image.

In using convolutional Neural Networks, I experimented with different architectures. I started by considering a simple CNN and then, to improve performace I built a Residual network experimenting with different architectures.

5.1. Simple CNN

At first, I considered a simple, funnel shaped convolutional neural network. In this network, the dimensionality of the image is decreased in two ways: with convolutional layers given the absence of padding, and with average pooling in 2x2 windows.

With this simple architecture, I went on to fine tune the learning rate and was able to obtain a validation accuracy of 13.76%.

In figure 5.1 we can see the results of the simple cnn experiment. On the two leftmost graphs we can see the progression of training and validation accuracy. Clearly, the training accuracy tends to go beyond validation accuracy when the model starts to overfit. However, in the training loop, I introduced an early stopping check that stops training when validation stops improving. This way, we see how overfitting is controlled. In fact, training and validation accuracy remain similar.

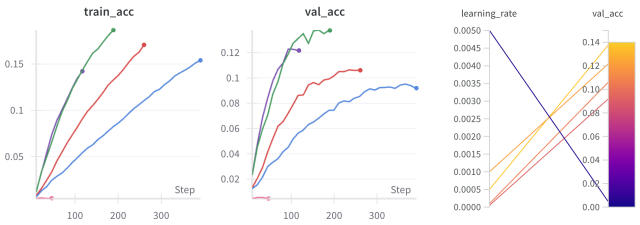

Figure 8. Simple CNN Experiment

In the rightmost picture, we see a parallel coordinates chart.

From the parallel coordinate chart we see that 4 out of 5 runs were fairly successful. The best run used a learning rate of 0.0005.

5.2. Resnet

To try to increase the performances of my system, I considered a residual network architecture. To implement it, I followed the explanation from the course textbook [1].

In experimenting with the Resnet architecture, I considered two different types of residual blocks: normal and bottleneck (depicted in figure 9). A normal Resnet block is composed of two applications of: a batch normalisation to stabilise training, a ReLu activation and a 3x3 convolutional layer. Notably, the output of the block is added back to its input, a technique known as skip connection. This feature improves the characteristics of the loss landscape and allows us to increase the depth of our networks.

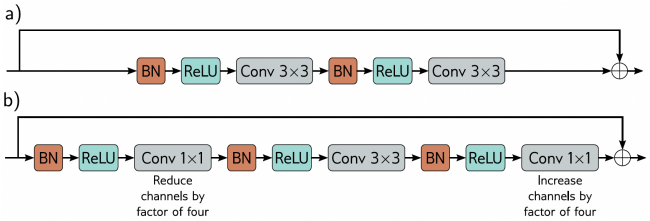

Figure 9. Resnet Blocks

Furthermore, I experimented with different numbers of pooling layers (4,5, and 6). As for the number of blocks, I settled for 20 in total. From the literature, I saw that some popular architectures use a similar number of blocks. Furthermore, I had performed an experiment where I tried to increase the number to 30 but the performances decreased.

In figure 10 we can see the results of the experiment on the ResNet architecture. All in all we obtain satisfactory performances with the normal block irrespectively of the number of pooling layers. For this reason, as the number of network parameters greatly decreases with pooling, I chose the model with normal blocks and 6 pooling layers as the best ResNet model. It had a validation accuracy around 20%.

Having chosen the best ResNet model, I went on and finetuned its hyperparameters to obtain better performances. I considered different batch sizes (16, 32, 64, 128, 192)

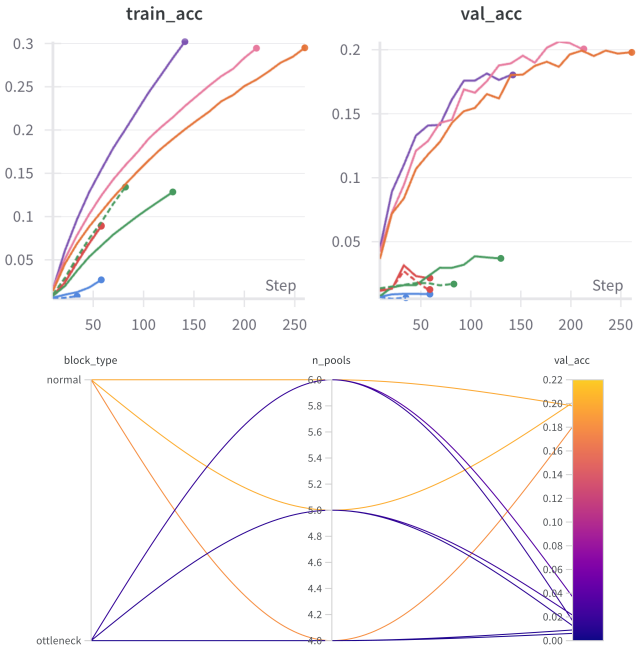

Figure 10. ResNet Experiment

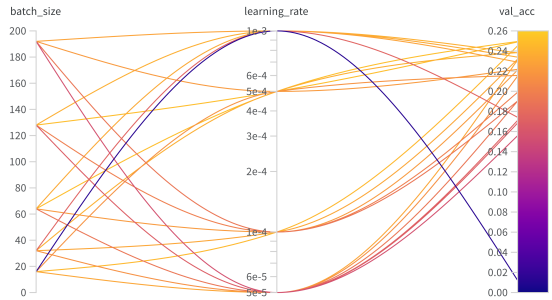

Figure 11. Final Resnet Hyperparameter Finetuning

and different learning rates (5e-5, 1e-4, 5e-4, 1e-3). In total, I performed 20 additional training runs and improved the validation accuracy to 24.95%. The resulting model used a batch size of 64 and a learning rate of 5e-4.

5.3. Best CNN Model Evaluation

Once I identified the best performing model, I downloaded its checkpoint and went on to evaluate its performance on the test set. As a result, in figure 12, I surprisingly obtained better performances than on validation. Since this result looked a bit suspicious to me, I checked that there was no data leaks from training.

Test accuracy	0.2856
Test f1 score	0.2762
Test precision	0.3047
Test recall	0.2856

Figure 12. Evaluation Metrics for the Best ResNet Model

Test accuracy	0.1828
Test f1 score	0.1702
Test precision	0.1828
Test recall	0.1661

Figure 13. Mediocre Model's Performance

5.4. Best CNN Model Exploration

As a last step I took the chance to explore the weights of my model. Actually, I had designed my CNNs having an initial convolution with a 7x7 kernel precisely for this reason. Recently, I have listened to a podcast on Mechanistic Interpretability [2] where it was sustained that Gabor filters represent a sort of "instrumental convergence". More specifically, it was said that vision models tend to develop Gabor filters in their early layers. I wanted to see if this fascinating hypothesis could be seen in my project. Therefore, I went on and plotted the first layer kernels of my best performing model as well as those of a mediocre one (for which I will post test performances in figure 13). The results can be seen in figures 14 and 15. We can confidently say that the best model's kernels look remarkably less random than the mediocre's ones. As for Gabor filters, they are not yet represented in the best model kernels. However, with some speculation, it is possible to notice a tendency for orientation selectivity as well as a tendency to use complementary colors to detect shapes. It would be interesting to see the dynamical evolution of the weights in a big SSL experiment.

6. Conclusions

In conclusion, I performed an Analysis to compare the performance of different methods for image classification. First, I considered different traditional classifiers on a SIFT + BoW representation. This approach, also because of computational limitations on my side, was not enough to

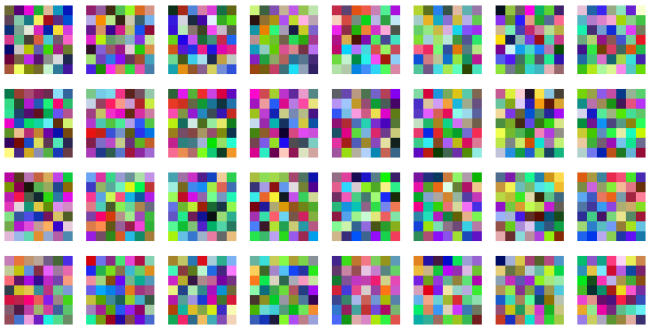

Figure 14. Mediocre Model's Kernels in the First Layer

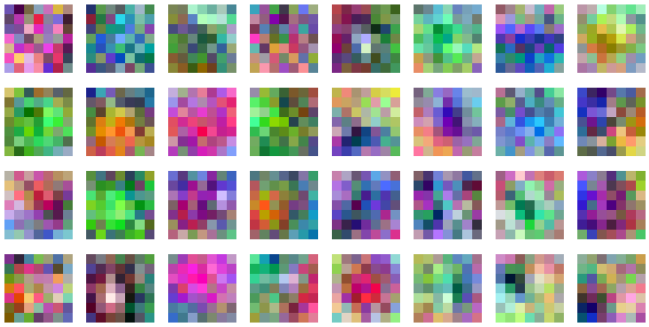

Figure 15. Best Model's Kernels in the First Layer

obtain decent results. Then, I considered a Deep Learning approach. I built different types of convolutional neural networks that ultimately yielded some decent results. From this experiment, and in this setting, the superiority of the deep learning approach with respect to the traditional one is clear.

7. Disclosure

I confirm that this report is entirely original and does not contain any form of plagiarism. I did not make us of ChatGPT or any other Natural Language Processing models.

References

[1] Simon J.D. Prince, *Understanding Deep Learning*, MIT Press, 2023. [https://udlbook.github.io/udlbook/]

[2] *mechint*: Mechanistic Interpretability explained [https://www.youtube.com/watch?v=riniamTdUSo]

[3] *lightning*: Pytorch Lightning [https://lightning.ai/pytorch-lightning]

[4] *wandb*: Weight and Biases [https://wandb.ai/site/]

[5] *opencv*: open computer vision library [https://opencv.org/]

[6] *opencv*: open computer vision library [https://opencv.org/]

[7] *scipy*: scientific computing in python [https://pypi.org/project/scipy/]

[8] *scikit-learn*: machine learning in python [https://scikit-learn.org/]

[9] *PyTorch*: An open-source machine learning framework [https://pytorch.org/].

[10] Lab Sessions: I have adapted some code and some approaches from the lab sessions.

[11] Course Slides: Apart from textbooks, I've used the slides of the course to choose and understand appropriate approaches.