[[1]](#footnote-1)

Iteration on the Fruit-360Dataset

Francesco Frassineti [fraf@itu.dk](mailto:fraf@itu.dk)

*Abstract*—In this paper I iterate o the work done on the Fruit-360 dataset available on Kaggle () to create a classifier of fruit and vegetables using convolutional neural networks. My project introduces improvements on the models proposed by the authors by trying new tunings of the parameters and by adding dropout layers between the fully connected layers of the Neural Network. I also train a model starting from the pre-trained model based on MobileNet V2. The results of the classifiers are then compared to the ones of the original paper and to a model based on logistic regression.

# INTRODUCTION

## The Dataset

T

he dataset used in this project is called Fruit-360 and can be downloaded from the addresses pointed by references. Currently, the dataset contains 90483 images of 131 different fruits and vegetables. At the time of writing *Mureșan, H., & Oltean, M. (2018)*, only 82213 images of 120 fruits and vegetables were available. The authors invite the reader to access the latest version of the dataset from the above indicated addresses.

The images were obtained by filming the fruits while they are rotated by a motor and then extracting the frames. A white sheet of paper was placed behind the fruits as background. Further work has been put to make sure the background was independent of the lighting conditions. Finally, fruits were scaled to fit a 100x100 pixels image. Each image contains one and only one fruit.

The dataset is already split between a training set (67692 images) and a test set (22688 images). The folder structure is the following:

* Images
  + Training
    - Apple Braeburn
    - Apple Crimson Snow
    - …
    - Watermelon
  + Test
    - Apple Braeburn
    - Apple Crimson Snow
    - …
    - Watermelon

## Motivation and Applications

I chose this dataset because I was interested in applying what I studied about Deep Learning to a real-life scenario starting from good quality data so that I could focus on the implementation, tuning and training of the machine learning model. Specifically, I wanted to work on Convolutional Neural Networks (CNNs) as they currently are the state-of-the-art classes of algorithms for image classification and detection. I also wanted to experiment with transfer learning, so I decided to train a network from scratch and compare it to a network that I could train from a pre-trained lightweight model such as MobileNet V2. I chose MobileNet V2 for its small size yet good performance, as I wanted my models to be small enough to work on mobile devices. Lastly, I decided to work on this dataset because, by reading *Mureșan, H., & Oltean, M. (2018)*, I realized that the authors did not use certain techniques in their model architecture that are known to improve generalization, so I wanted to see if by introducing them there would be improvements in the performance of the classifiers. Specifically, I’m referring to adding Dropout layers between each couple of consecutive fully connected layers.

My work may be applied across multiple domains. For example, the trained models could be inserted into a portable device to be used by visually impaired people to get help to recognize between different fruits and vegetables. It may also be applied to autonomous fruit harvesting in greenhouses or to the identification of out of place items in the aisles of stores.

# Theory

In this section I will briefly explain the key aspects of the theory used to back my work. For the sake of brevity, I will try to focus on the aspects of deep learning that are related to CNNs and image classification.

## Convolutional Neural Networks

Convolutional Neural Networks are a specialized kind of neural network for processing data with a grid-like topology (Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.). The most common application on CNNs is image classification and detection. Typical CNN architectures follow the following pattern:

* Sequence of Convolutional Layers (with ReLU as the activation function) and Pooling Layers one after the other
* Flatten Layer
* Sequence of Fully Connected Layers

Compared to fully connected NNs, CNNs take knowledge on the topology of the data into consideration and this allows the number of trainable parameters necessary to get the same performance of a traditional NN to be much lower.

### Convolutional Layer

Convolutional Layers are named after the convolution operation. A convolutional layer consists of groups of neurons that make up kernels. The kernels have a small size and they slide across the width and height of the input, extract high level features and produce a 2-dimensional activation map to be used as the input of the following layer.

### Pooling Layers

Pooling layers are used to:

1. Reduce the spatial dimensions of the representation
2. Reduce the amount of computation done in the network
3. Control overfitting

### Flatten Layer

Converts the output of the convolutional part of the CNN from a 2D to 1D representation so that it can be fed to the fully connected part.

### Fully Connected Layers

Each neuron from a fully connected layer is linked to each output of the previous layer.

## Dropout

Machine learning models suffer from overfitting when they achieve good classification results on the training set, but the model doesn’t manage to generalize well on the test set.

Overfitting is a very common problem when training a classifier and *Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014)* proposed Dropout as a possible way to reduce it. It’s a technique that has been proven to greatly improve the property of generalization in NN-based models. Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored or “dropped out.” This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different “view” of the configured layer. Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.

This conceptualization suggests that perhaps dropout breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust. As a rule of thumb, when introducing dropout to the network, it’s suggested to double the amount of neurons in the layer and use a *dropout-rate* of 0.5 for hidden layers and 0.8 for input layers.

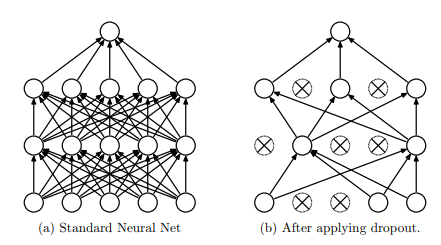


Figure : Dropout

## ReLU

ReLU djfhsjdfhsdfjshdbf

# The Method

In this section I write about the python implementation of the 3 different types of classifier that I trained with Keras:

* The CNNs trained from scratch, referred as Model As
* The CNNs trained from MobileNet V2, referred as Model Bs
* Logistic Regression as the baseline algorithm (Model C)

I will use the python code I wrote as a reference, but I will not quote it so not to decrease readability. The boilerplate for the 3 types of classifier is almost the same, so I discuss it step by step and I point out the specific differences when necessary.

## Libraries

These are the required libraries to run the project:

* *Numpy*: fundamental package for scientific computing with Python.
* *Tensorflow*: an end-to-end open source machine learning platform.
* *Keras*: open-source neural-network library written in Python. It allows to write deep learning model architectures at higher level compared to Tensorflow.
* *Matplotlib.pyplot*: library for plotting charts.
* *Datetime*: library to handle dates and time.
* *Os*: library to do operations at a operative system level.

## Preprocessing: Data Augmentation and Normalization

In order to improve the quality and the quantity of the data, the images flowing from the training and testing directory went through the following steps:

* *Normalization*: in the *ImageDataGenerator* constructors, the *rescale* parameter has been set to 1./255, therefor rescaling every value of every channel from [0; 255] to [0;1]. This is meant to improve the quality of the stochastic gradient descent.
* *Augmentation*: in the *ImageDataGenerator* of the training (and validation) set, the following parameters have been specified:
  + *Shear\_range=0.1*: add random shear to the training example
  + *Zoom\_range=0*.1: add random zoom
  + *Horizontal\_flip*=True: flip the image with a probability of 50%.

The last step of the preprocessing consisted in splitting the training set into the actual training set (80% of the training images) and a validation set (20%) to be used to get an estimate of the performance of the classifiers during training.

The output of the preprocessing step consists in the following 3 types of batches (*batch\_size=50)*:

* Train\_batches
* Validation\_batches
* Test\_batches

## Definition of the Model Architecture

### CNN trained from Scratch (Model A)

Model A is a sequential model where each convolutional layer has *kernel\_size=3x3* and uses *valid padding* (= no padding). Each MaxPooling layer has *stride=2* and *pool\_size=(2x2)* therefor decreasing the image area by 4 times. The chosen activation function for every neuron in a hidden layer is the *Rectified Linear Unit* (ReLU), while for the output layer it’s ‘softmax’ as we are training a multi-class classifier. Now let’s focus on the actual topology of the network:

* Conv2D Layer (16 output features)
* MaxPooling2D
* Conv2D Layer (32 output features)
* MaxPooling2D
* Conv2D Layer (64 output features)
* MaxPooling2D
* Conv2D Layer (128 output features)
* MaxPooling2D
* Conv2D Layer (256 output features)
* Flatten()
* Dense Layer (2048 neurons)
* Dropout Layer (*dropout\_rate=0.5*)
* Dense Layer (512 neurons)
* Dropout Layer (*dropout\_rate=0.5*)
* Dense Layer (131 output neurons)

*Resulting Model Size*: 42.361 MB

*Number of Parameters*: 3608099

### CNN trained from MobileNet V2 (Model B)

Model B started from MobileNet V2, a pretrained model included in *tensorflow.keras.applications*. The model has been loaded with the following parameters: *input\_shape=(100, 100, 3)* so that it matched the size of the images in the Fruit-360 dataset. The starting weights are the ones calculated by training on the ‘*imagenet’* dataset. The model is specified to perform *max pooling* as the preferred type of pooling.

Lastly, *include\_top* is set to *False* so that I could drop the pre-trained output layer and add my custom layers on top of the pretrained model to adapt the model to predict my 131 classes. These were the layers that I added to the pre-trained model:

* Dense Layer (2048 neurons, *ReLU*)
* Dropout (*dropout\_rate=0.5*)
* Dense Layer(131 output neurons, *softmax*)

*Resulting Model Size*: 60.929 MB

*Number of Parameters*: 5149891

### Logistic Regression (Model C)

Model C simply consisted in a model based on logistic regression in order to have a comparison with a baseline algorithm. The keras model is the following:

* Flatten layer
* Dense layer (131 output layers, *softmax*)

*Resulting Model Size*: 46.074 MB

*Number of Parameters*: 3939131

## Model Compilation

The models were compiled by using an *Adam Optimizer* with default parameters and variable *learning\_rate*. Since multi-label classification was the goal, the loss function of choice was *categorical\_crossentropy*. The model was compiled to evaluate both *loss* and *accuracy.*

## Model Training

Before starting the actual training, 2 callbacks to be triggered at the end of each training epoque were defined:

* *EarlyStopping*: this callback will stop training when the chosen performance measure (*validation accuracy*) stops improving. Because of the EarlyStopping callback, I could specify a large amount of training epochs and, by setting *patience=25*, I could expect the model to stop training in a reasonable time after the network stopped improving.
* *ModelCheckpoint*: I set this callback to save model with the best *validation accuracy* at the end of each epoque.

Once the two callbacks are defined, I start the training by feeding them to the fit function together with x*=training\_batches, validation\_data=validation\_batches*.

The history of the training is memorized in the *history* variable and the time elapsed between the start and the end of the training is memorized in the *time\_delta* variable.

After the training, the last iteration of the classifier and the one with the best *validation accuracy* can be found in the *models/<type>/<date\_time>* folder. On top of that, a text file with the summary of the model architecture, its hyper-parameters and training information is saved in the same folder.

Hyper-parameters include:

* Learning Rate
* Dropout Rate
* Batch Size

Training information includes:

* Time Train Start
* Time Train End
* Time Train Delta
* Train Accuracy
* Validation Accuracy

Lastly, the following two charts are saved:

* Loss history (train and validation)
* Accuracy history (train and validation)

## Evaluation of the Classifier

After the training, I proceed to load the model with the best *validation accuracy* and evaluate its accuracy against the test set. The evaluated metric is then appended to the previously mentioned summary file.

# Numerical Experiments and Results

In order to try to achieve the best test accuracy, I trained training models experimenting with different hyper-parameters. Specifically, I trained models with different learning rates. *Table 1* shows the results of training models of type A (non pre-trained CNN with Dropout), B (pre-trained CNN from MobileNet V2 with Dropout) and C (Logistic Regression). The results are compared with the 3 best classifiers (Prev1, Prev 7 and Prev2) from *Mureșan, H., & Oltean, M. (2018)*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | Learning Rate | Train Accuracy | Test Accuracy | Train Time (s) |
| A1 | 5e-4 | 99.89% | 94.87% | 7823.95 |
| A2 | 3e-4 | 99.919% | 96.756% | 6269.54 |
| A3 | 1e-4 |  |  |  |
| A4 | 5e-5 | 99.9963% | 96.93% | 7753.23 |
| A5 | 1e-5 | 99.9982% | 96.55% | 11580.2 |
| B1 | 1e-3 | 99.56% | 96.28% | 10462 |
| B2 | 1e-4 |  |  |  |
| C1 | 5e-2 | 97.07% | 88.11% | 10961.2 |
| Prev1 | - | 99.58% | 95.23% | - |
| Prev7 | - | 99.55% | 95.09% | - |
| Prev2 | - | 99.68% | 95.02% | - |

Table :results of the training sessions of the different types of classifiers and different learning rates compared with each other and with the previous work by Mureșan, H., & Oltean, M. (2018).

By choosing appropriate learning rates, the CNNs with droupout that I trained tend to perform slightly better compared to the ones without dropout by Mureșan, H., & Oltean, M. (2018). There is still some overfitting, as the train accuracies tend to 100% while the test accuracies never reach 97%. As expected, the lower the learning rate, the less overshooting is performed during gradient descent, making the training loss more stable. For model A1 and B1 the learning rate is too high and that causes the validation loss to increase over time as a sign of overfitting. The model that suffers most from overfitting, though, is C1, the one based on logistic regression: while the model acceptably fits the training data (training accuracy is 97%), the model highly suffers of overfitting as the test accuracy is barely 88.11%. This was expected, as logistic regression models are conceptually very simple:

1. There is only one output layer fully connected to the input layer without any hidden layer, thus decreasing the abstraction capabilities of the model.
2. It does not exploit the grid-like structure of the input data.

That’s why model C1 performed much worse compared to models of type A even if they all had comparable parameter numbers.

TRAINING TIMES

|  |  |  |  |
| --- | --- | --- | --- |
|  | Learning Rate | Loss History | Accuracy History |
| A1 | 5e-4 |  |  |
| A2 | 3e-4 |  |  |
| A3 | 1e-4 | sdasdasd | asdasdas |
| A4 | 5e-5 |  |  |
| A5 | 1e-5 |  |  |
| B1 | 1e-3 |  |  |
| B2 | 1e-4 | asdasdasd | asdasdasda |
| C1 | 5e-2 |  |  |

Table : Histories of Loss and Accuracy on the Training and **Validation** Set (Errata Corrige: in the legends of the charts I mistakenly wrote “test” instead of “validation”)

# Discussion

Hsdfjshdfj

# Conclusion

References

1. Mureșan, H., & Oltean, M., “Fruit recognition from images using deep learning” (2018).

1. [↑](#footnote-ref-1)