Iteration on the Fruit-360 Dataset

Francesco Frassineti fraf@itu.dk

***Abstract*—In this paper I iterate on the work done by *Mureșan, H., & Oltean, M. (2018)* on the Fruit-360 dataset available at https://www.kaggle.com/moltean/fruits 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.**

I. Introduction

A. *The Dataset*

THE dataset used in this project is called Fruit-360 and can be downloaded from www.kaggle.com/moltean/fruits. 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 address indicated above.

The images were obtained by filming fruits and vegetables while they were being rotated by a motor and by extracting the frames afterwards. 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
-

B. *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 models. 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 recognized as useful 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.

II. Theory

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

A. *Convolutional Neural Networks*

Convolutional Neural Networks are a specialized kind of neural network for processing data with a grid-like topology (Goodfellow, Bengio, & Courville, A. (2016)). The most common applications of CNNs are image classification and detection. Typical CNN architectures consist of the following:

- Sequence of Convolutional Layers (with *ReLU* as the activation function) and Pooling Layers alternated one after the other.
- Flatten Layer: it converts the dimensionality of the input data from 2D to 1D.
- Sequence of Fully Connected Layers.

Compared to fully connected NNs, CNNs take knowledge on the topology of the data into consideration, therefor improving the training performance.

1) *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.

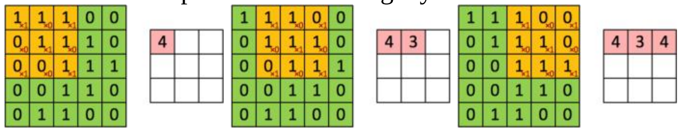


Figure 1: Example of the Convolution between an image and a 3x3 Kernel. Image from Frassineti (2017).

2) *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

A typical pooling operation in CNNs is *MaxPooling*: given a neighborhood of values in a feature map, the result of this calculation is their maximum value.

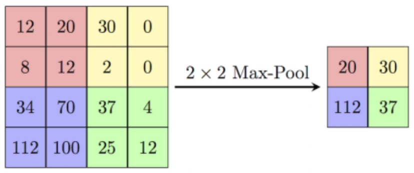


Figure 2: Example of MaxPooling operation. Image from https://computersciencewiki.org/index.php/Max-pooling_/_Pooling

3) *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.

4) *Fully Connected Layers*

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

B. *Rectified Linear Unit (ReLU)*

The Rectified Linear Unit is an activation function defined as $f(x) = x^+ = \max(0, x)$. Compared to *sigmoid* and other activation functions, *ReLU* is demonstrated to enable better training of deep neural networks (Glorot, Bordes, & Bengio (2011)) and it's the activation function of choice for the hidden layers of CNNs.

C. *Dropout*

Machine learning models suffer from overfitting when they achieve good classification results on the training set, but the model doesn't generalize well on the test set.
Overfitting is a very common problem when training a classifier and *Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov(2014)* proposed Dropout as a possible way to reduce it. It's a method 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 number of neurons in the layer and use a *dropout-rate* of 0.5 for hidden layers and 0.8 for input layers.

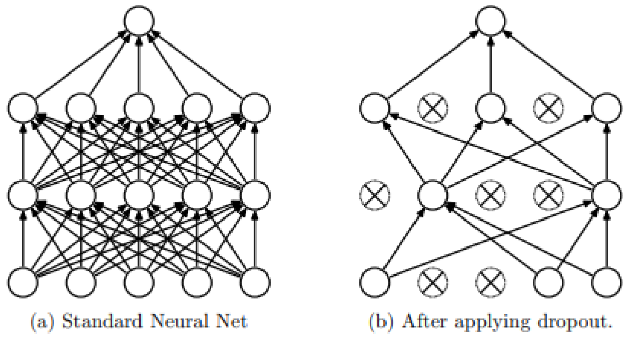


Figure 3: Dropout. From Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov(2014).

III. The Method

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

- The CNNs trained from scratch, referred as type A
- The CNNs trained from MobileNet V2, referred as type B
- The Logistic Regression model, referred as type 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.

A. *Libraries*

These are the required libraries to run the project:

- *Numpy*: package for scientific computing with Python.
- *Tensorflow*: the end-to-end open source machine learning platform by *Google*.
- *Keras*: open-source neural-network library that allows developers to write deep learning model architectures at a higher level compared to Tensorflow.
- *Matplotlib.pyplot*: library for plots and charts.
- *Datetime*: library to handle dates and time.
- *Os*: library to do operations at level of the operative system.

B. *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 color 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 horizontally 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

C. Definition of the Model Architecture

1) 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 size 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 multiclass 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 Layer
- 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

2) CNN trained from MobileNet V2 (Model B)

Model B starts from MobileNet V2, a pretrained model included in *tensorflow.keras.applications*. The model is loaded with the following parameters: *input_shape=(100, 100, 3)* so that it matches 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 the pre-trained output layer is replaced by new custom layers to allow the prediction of the 131 classes of fruits and vegetables. These are the layers that are 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

3) Logistic Regression (Model C)

Model C is a model based on logistic regression to be compared to models of type A and B. The Keras model is the following:

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

Resulting Model Size: 46.074 MB
Number of Parameters: 3939131

D. Model Compilation

The models are compiled by using an *Adam Optimizer* with default parameters and variable *learning_rate*. An *Adam Optimizer* performs Stochastic Gradient Descent with a learning rate that progressively decreases for each iteration. Since multi-class classification is the goal, the loss function of choice is *categorical_crossentropy*. The model is compiled to evaluate both *loss* and *accuracy* as metrics.

E. Model Training

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

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

Once the two callbacks are ready, I start the training session on a *GeForce GTX 1060* 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

Lastly, the following two charts are saved:

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

F. *Evaluation of the Classifier*

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

IV. Numerical Experiments and Results

A. *Previous Work*

Before writing about the actual results, let's briefly introduce the models from *Mureșan, H., & Oltean, M. (2018)*. They all are CNNs with a kernel size of 5x5. In order to differentiate them from the models trained in the current project, I will name them Z1, Z7 and Z2 respectively. The numbers are taken from the *"Nr."* column in *Table 4* of pag.23 of *Mureșan, H., & Oltean, M. (2018)*.

Model	**Configuration**		
Z1	Convolutional	5 x 5	16
	Convolutional	5 x 5	32
	Convolutional	5 x 5	64
	Convolutional	5 x 5	128
	Fully Connected	-	1024
	Fully Connected	-	256
Z7	Convolutional	5 x 5	16
	Convolutional	5 x 5	32
	Convolutional	5 x 5	128
	Convolutional	5 x 5	128
	Fully Connected	-	1024
	Fully Connected	-	256
Z2	Convolutional	5 x 5	8
	Convolutional	5 x 5	32
	Convolutional	5 x 5	64
	Convolutional	5 x 5	128
	Fully Connected	-	1024
	Fully Connected	-	256

Table 1: Top-3 models from Mureșan, H., & Oltean, M. (2018).

B. *Results*

In order to try to achieve the best test accuracy, models were trained by experimenting with different hyper-parameters. Specifically, models were trained with different learning rates. *Table 1* shows the results of training models of type A (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 (*Z1, Z7* and *Z2*) 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	99.942%	95.363%	5594.06
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	99.878%	98.572%	11994.77
B3	3e-5	100%	98.766%	8885.65
C1	5e-2	97.07%	88.11%	10961.2
Z1	-	99.58%	95.23%	-
Z7	-	99.55%	95.09%	-
Z2	-	99.68%	95.02%	-

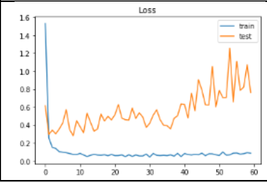
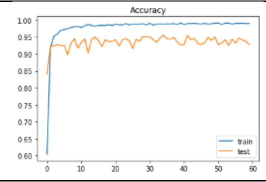
Table 2:results of the training sessions of the different types of classifiers and different learning rates compared with each other and with the previous work by Mureșan, H., & Oltean, M. (2018).

By choosing appropriate learning rates, the CNNs with dropout layers tend to perform slightly better compared to the ones without them 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% with type A. As expected, the lower the learning rate, the less overshooting is performed during gradient descent, making the training loss more stable and improving generalization. For models A1-4 and B1-2 the learning rates are too high and that causes the validation loss to increase over time as a sign of overfitting. The model that suffers most from overfitting 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.

B1 has a good performance that is comparable with the one of models of type A. By looking at its loss and accuracy history (*table 3*), though, it is clear that a better model can be achieved by trying a lower *learning rate*. By choosing a *learning rate=1e-4*, we obtain model B2. By looking at its loss and accuracy history, it can be observed that the *learning rate* can be lowered further (e.g.: to 3e-5). The resulting model B3 achieves the best classification performance out of all the considered models with a *test accuracy of 98.766%*.

	Learning Rate	Loss History	Accuracy History
A 1	5e-4		

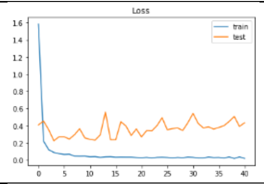
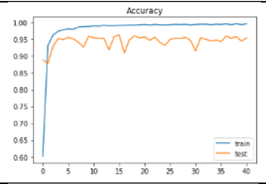
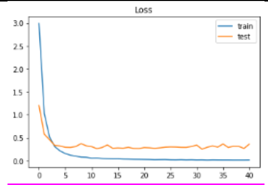
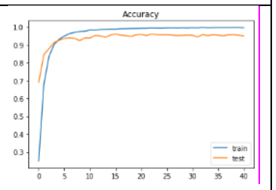
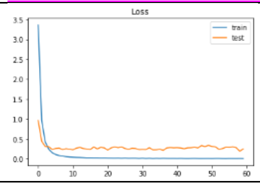
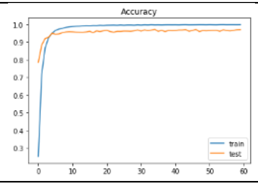
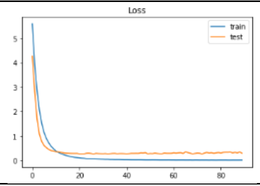
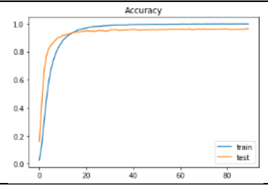
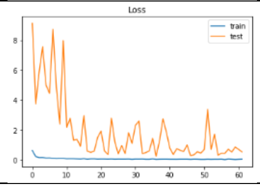
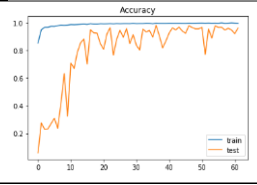
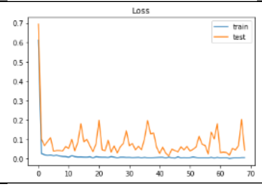
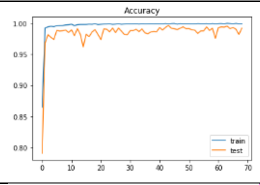
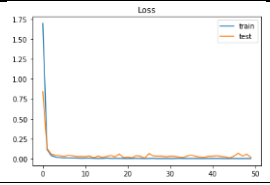
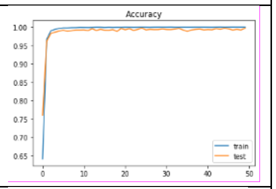
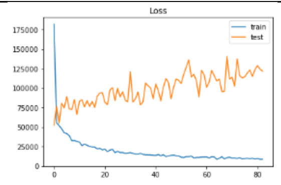
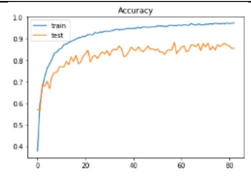
A 2	3e-4		
A 3	1e-4		
A 4	5e-5		
A 5	1e-5		
B1	1e-3		
B2	1e-4		
B3	3e-5		
C1	5e-2		

Table 3: 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")*

V. Discussion

My results showed that improvements over the previously proposed classification models are possible and that adding dropout and starting from a pre-trained model are both good ideas to try to achieve higher classification accuracy. Therefore, I consider my experiment to have had a successful outcome. Due to the fact that the Fruit-360 dataset was created starting from frames of videos of objects spinning around a specific axis, I think further investigation should be done to see if these classifiers and the models proposed by Mureșan, H., & Oltean, M. (2018) can actually generalize to any kind of photo of a single piece of fruit or vegetable, regardless of the rotation axis.

A. *Privacy Aspect*

Since the dataset is composed of pictures of inanimate objects and at no point users of future applications are supposed to be asked about their personal data related to the dataset itself, I don't see how my project might raise concerns about privacy on its own. If the fruit recognition classifier is combined with a face recognition model or another kind of personal identification system, it could be possible for a third party to collect information about the habits of their customer and privacy concerns may assume relevancy.

References

- Mureșan, H., & Oltean, M. (2018). Fruit recognition from images using deep learning.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.
- Glorot, X., Bordes, A., & Bengio, Y. (2011, June). Deep sparse rectifier neural networks. In Proceedings of the fourteenth international conference on artificial intelligence and statistics (pp. 315-323).
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.