Fruit recognition from images using deep learning

Horea Mureşan
Faculty of Mathematics and Computer
Science
Mihail Kogălniceanu, 1
Babeş-Bolyai University
Romania
email: horea94@gmail.com

Mihai Oltean
Faculty of Exact Sciences and
Engineering
Unirii, 15-17
"1 Decembrie 1918" University of Alba
Iulia
Romania
email: mihai.oltean@gmail.com

Abstract.

In this paper we introduce a new, high-quality, dataset of images containing fruits. We also present the results of some numerical experiment for training a neural network to detect fruits. We discuss the reason why we chose to use fruits in this project by proposing a few applications that could use such classifier.

Keywords: *Deep learning*, *Object recognition*, *Computer vision*, *fruits dataset*, *image processing*

1 Introduction

The aim of this paper is to propose a new dataset of images containing popular fruits. The dataset was named Fruits-360 and can be downloaded from the addresses pointed by references [18] and [19]. Currently (as of 2019.01.22) the set contains 65429 images of 95 fruits and it is constantly updated with images of new fruits as soon as the authors have accesses to them. The reader is encouraged to access the latest version of the dataset from the above indicated addresses.

1

Having a high-quality dataset is essential for obtaining a good classifier. Most of the existing datasets with images (see for instance the popular CIFAR dataset [12]) contain both the object and the noisy background. This could lead to cases where changing the background will lead to the incorrect classification of the object.

As a second objective we have trained a deep neural network that is capable of identifying fruits from images. This is part of a more complex project that has the target of obtaining a classifier that can identify a much wider array of objects from images. This fits the current trend of companies working in the augmented reality field. During its annual I/O conference, Google announced [21] that is working on an application named Google Lens which will tell the user many useful information about the object toward which the phone camera is pointing. First step in creating such application is to correctly identify the objects. The software has been released later in 2017 as a feature of Google Assistant and Google Photos apps. Currently the identification of objects is based on a deep neural network [35].

Such a network would have numerous applications across multiple domains like autonomous navigation, modeling objects, controlling processes or human-robot interactions. The area we are most interested in is creating an autonomous robot that can perform more complex tasks than a regular industrial robot. An example of this is a robot that can perform inspections on the aisles of stores in order to identify out of place items or under-stocked shelves. Furthermore, this robot could be enhanced to be able to interact with the products so that it can solve the problems on its own. Another area in which this research can provide benefits is autonomous fruit harvesting. While there are several papers on this topic already, from the best of our knowledge, they focus on few species of fruits or vegetables. In this paper we attempt to create a network that can classify a variety of species of fruit, thus making it useful in many more scenarios.

As the start of this project we chose the task of identifying fruits for several reasons. On one side, fruits have certain categories that are hard to differentiate, like the citrus genus, that contains oranges and grapefruits. Thus we want to see how well can an artificial intelligence complete the task of classifying them. Another reason is that fruits are very often found in stores, so they serve as a good starting point for the previously mentioned project.

The paper is structured as follows: in the first part we will shortly discuss

a few outstanding achievements obtained using deep learning for fruits recognition, followed by a presentation of the concept of deep learning. In the second part we describe the Fruits-360 dataset: how it was created and what it contains. In the third part we will present the framework used in this project - TensorFlow[32] and the reasons we chose it. Following the framework presentation, we will detail the structure of the neural network that we used. We also describe the training and testing data used as well as the obtained performance. Finally, we will conclude with a few plans on how to improve the results of this project. Source code is listed in the Appendix.

2 Related work

In this section we review several previous attempts to use neural networks and deep learning for fruits recognition.

A method for recognizing and counting fruits from images in cluttered greenhouses is presented in [28]. The targeted plants are peppers with fruits of complex shapes and varying colors similar to the plant canopy. The aim of the application is to locate and count green and red pepper fruits on large, dense pepper plants growing in a greenhouse. The training and validation data used in this paper consists of 28000 images of over 1000 plants and their fruits. The used method to locate and count the peppers is two-step: in the first step, the fruits are located in a single image and in a second step multiple views are combined to increase the detection rate of the fruits. The approach to find the pepper fruits in a single image is based on a combination of (1) finding points of interest, (2) applying a complex high-dimensional feature descriptor of a patch around the point of interest and (3) using a so-called bag-of-words for classifying the patch.

Paper [25] presents a novel approach for detecting fruits from images using deep neural networks. For this purpose the authors adapt a Faster Region-based convolutional network. The objective is to create a neural network that would be used by autonomous robots that can harvest fruits. The network is trained using RGB and NIR (near infra red) images. The combination of the RGB and NIR models is done in 2 separate cases: early and late fusion. Early fusion implies that the input layer has 4 channels: 3 for the RGB image and one for the NIR image. Late fusion uses 2 independently trained models that are merged by obtaining predictions from both models and averaging the results. The result is a multi modal network

which obtains much better performance than the existing networks.

On the topic of autonomous robots used for harvesting, paper [1] shows a network trained to recognize fruits in an orchard. This is a particularly difficult task because in order to optimize operations, images that span many fruit trees must be used. In such images, the amount of fruits can be large, in the case of almonds up to 1500 fruits per image. Also, because the images are taken outside, there is a lot of variance in luminosity, fruit size, clustering and view point. Like in paper [25], this project makes use of the Faster Region-based convolutional network, which is presented in a detailed view in paper [24]. Related to the automatic harvest of fruits, article [22] presents a method of detecting ripe strawberries and apples from orchards. The paper also highlights existing methods and their performance.

In [11] the authors compile a list of the available state of the art methods for harvesting with the aid of robots. They also analyze the method and propose ways to improve them.

In [2] one can see a method of generating synthetic images that are highly similar to empirical images. Specifically, this paper introduces a method for the generation of large-scale semantic segmentation datasets on a plant-part level of realistic agriculture scenes, including automated per-pixel class and depth labeling. One purpose of such synthetic dataset would be to bootstrap or pre-train computer vision models, which are fine-tuned thereafter on a smaller empirical image dataset. Similarly, in paper [23] we can see a network trained on synthetic images that can count the number of fruits in images without actually detecting where they are in the image.

Another paper, [4], uses two back propagation neural networks trained on images with apple "Gala" variety trees in order to predict the yield for the upcoming season. For this task, four features have been extracted from images: total cross-sectional area of fruits, fruit number, total cross-section area of small fruits, and cross-sectional area of foliage.

Paper [10] presents an analysis of fruit detectability in relation to the angle of the camera when the image was taken. Based on this research, it was concluded that the fruit detectability was the highest on front views and looking with a zenith angle of 60° upwards.

In papers [27, 37, 15] we can see an approach to detecting fruits based on color, shape and texture. They highlight the difficulty of correctly classifying similar fruits of different species. They propose combining existing methods using the texture, shape and color of fruits to detect regions of interest from

images. Similarly, in [20] a method combining shape, size and color, texture of the fruits together with a k nearest neighbor algorithm is used to increase the accuracy of recognition.

One of the most recent works [36] presents an algorithm based on the improved ChanVese level-set model [3] and combined with the level-set idea and M-S mode [17]. The proposed goal was to conduct night-time green grape detection. Combining the principle of the minimum circumscribed rectangle of fruit and the method of Hough straight-line detection, the picking point of the fruit stem was calculated.

3   Deep learning

In the area of image recognition and classification, the most successful results were obtained using artificial neural networks [6, 30]. These networks form the basis for most deep learning models.

Deep learning is a class of machine learning algorithms that use multiple layers that contain nonlinear processing units [26]. Each level learns to transform its input data into a slightly more abstract and composite representation [6]. Deep neural networks have managed to outperform other machine learning algorithms. They also achieved the first superhuman pattern recognition in certain domains [5]. This is further reinforced by the fact that deep learning is considered as an important step towards obtaining Strong AI. Secondly, deep neural networks - specifically convolutional neural networks - have been proved to obtain great results in the field of image recognition.

In the rest of this section we will briefly describe some models of deep artificial neural networks along with some results for some related problems.

3.1   Convolutional neural networks

Convolutional neural networks (CNN) are part of the deep learning models. Such a network can be composed of convolutional layers, pooling layers, ReLU layers, fully connected layers and loss layers [34]. In a typical CNN architecture, each convolutional layer is followed by a Rectified Linear Unit (ReLU) layer, then a Pooling layer then one or more convolutional layer and finally one or more fully connected layer. A characteristic that sets apart the CNN from a regular neural network is taking into account the structure

of the images while processing them. Note that a regular neural network converts the input in a one dimensional array which makes the trained classifier less sensitive to positional changes.

Among the best results obtained on the MNIST [13] dataset is done by using multi-column deep neural networks. As described in paper [7], they use multiple maps per layer with many layers of non-linear neurons. Even if the complexity of such networks makes them harder to train, by using graphical processors and special code written for them. The structure of the network uses winner-take-all neurons with max pooling that determine the winner neurons.

Another paper [16] further reinforces the idea that convolutional networks have obtained better accuracy in the domain of computer vision. In paper [29] an all convolutional network that gains very good performance on CIFAR-10 [12] is described in detail. The paper proposes the replacement of pooling and fully connected layers with equivalent convolutional ones. This may increase the number of parameters and adds inter-feature dependencies however it can be mitigated by using smaller convolutional layers within the network and acts as a form of regularization.

In what follows we will describe each of the layers of a CNN network.

3.1.1 Convolutional layers

Convolutional layers are named after the convolution operation. In mathematics convolution is an operation on two functions that produces a third function that is the modified (convoluted) version of one of the original functions. The resulting function gives in integral of the pointwise multiplication of the two functions as a function of the amount that one of the original functions is translated [33].

A convolutional layer consists of groups of neurons that make up kernels. The kernels have a small size but they always have the same depth as the input. The neurons from a kernel are connected to a small region of the input, called the receptive field, because it is highly inefficient to link all neurons to all previous outputs in the case of inputs of high dimensions such as images. For example, a 100 x 100 image has 10000 pixels and if the first layer has 100 neurons, it would result in 1000000 parameters. Instead of each neuron having weights for the full dimension of the input, a neuron holds weights for the dimension of the kernel input. The kernels slide across the width and height of the input, extract high level features and produce a 2 dimensional activation map. The stride at which a kernel slides is given

as a parameter. The output of a convolutional layer is made by stacking the resulted activation maps which in turned is used to define the input of the next layer.

Applying a convolutional layer over an image of size 32 X 32 results in an activation map of size 28 X 28. If we apply more convolutional layers, the size will be further reduced, and, as a result the image size is drastically reduced which produces loss of information and the vanishing gradient problem. To correct this, we use padding. Padding increases the size of a input data by filling constants around input data. In most of the cases, this constant is zero so the operation is named zero padding. "Same" padding means that the output feature map has the same spatial dimensions as the input feature map. This tries to pad evenly left and right, but if the number of columns to be added is odd, it will add an extra column to the right. "Valid" padding is equivalent to no padding.

The strides causes a kernel to skip over pixels in an image and not include them in the output. The strides determines how a convolution operation works with a kernel when a larger image and more complex kernel are used. As a kernel is sliding the input, it is using the strides parameter to determine how many positions to skip.

ReLU layer, or Rectified Linear Units layer, applies the activation function $max(0, x)$. It does not reduce the size of the network, but it increases its nonlinear properties.

3.1.2 Pooling layers

Pooling layers are used on one hand to reduce the spatial dimensions of the representation and to reduce the amount of computation done in the network. The other use of pooling layers is to control overfitting. The most used pooling layer has filters of size 2 x 2 with a stride 2. This effectively reduces the input to a quarter of its original size.

3.1.3 Fully connected layers

Fully connected layers are layers from a regular neural network. Each neuron from a fully connected layer is linked to each output of the previous layer. The operations behind a convolutional layer are the same as in a fully connected layer. Thus, it is possible to convert between the two.

3.1.4   Loss layers

Loss layers are used to penalize the network for deviating from the expected output. This is normally the last layer of the network. Various loss function exist: softmax is used for predicting a class from multiple disjunct classes, sigmoid cross-entropy is used for predicting multiple independent probabilities (from the [0, 1] interval).

3.2   Recurrent neural network

Another deep learning algorithm is the recursive neural network [16]. The paper proposes an improvement to the popular convolutional network in the form of a recurrent convolutional network. In this kind of architecture the same set of weights is recursively applied over some data. Traditionally, recurrent networks have been used to process sequential data, handwriting or speech recognition being the most known examples. By using recurrent convolutional layers with some max pool layers in between them and a final global max pool layer at the end several advantages are obtained. Firstly, within a layer, every unit takes into account the state of units in an increasingly larger area around it. Secondly, by having recurrent layers, the depth of the network is increased without adding more parameters. Recurrent networks have shown good results in natural language processing.

3.3   Deep belief network

Yet another model that is part of the deep learning algorithms is the deep belief network [14]. A deep belief network is a probabilistic model composed by multiple layers of hidden units. The usages of a deep belief network are the same as the other presented networks but can also be used to pre-train a deep neural network in order to improve the initial values of the weights. This process is important because it can improve the quality of the network and can reduce training times. Deep belief networks can be combined with convolutional ones in order to obtain convolutional deep belief networks which exploit the advantages offered by both types of architectures.

4   Fruits-360 data set

In this section we describe how the data set was created and what it contains.

The images were obtained by filming the fruits while they are rotated by a motor and then extracting frames.

Fruits were planted in the shaft of a low speed motor (3 rpm) and a short movie of 20 seconds was recorded. Behind the fruits we placed a white sheet of paper as background.

Figure 1: Left-side: original image. Notice the background and the motor shaft. Right-side: the fruit after the background removal and after it was scaled down to 100x100 pixels.

However due to the variations in the lighting conditions, the background was not uniform and we wrote a dedicated algorithm which extract the fruit from the background. This algorithm is of flood fill type: we start from each edge of the image and we mark all pixels there, then we mark all pixels found in the neighborhood of the already marked pixels for which the distance between colors is less than a prescribed value. we repeat the previous step until no more pixels can be marked.

All marked pixels are considered as being background (which is then filled with white) and the rest of pixels are considered as belonging to the object. The maximum value for the distance between 2 neighbor pixels is a parameter of the algorithm and is set (by trial and error) for each movie.

Fruits were scaled to fit a 100x100 pixels image. Other datasets (like MNIST) use 28x28 images, but we feel that small size is detrimental when

you have too similar objects (a red cherry looks very similar to a red apple in small images). Our future plan is to work with even larger images, but this will require much more longer training times.

To understand the complexity of background-removal process we have depicted in Figure 1 a fruit with its original background and after the background was removed and the fruit was scaled down to 100 x 100 pixels.

The resulted dataset has 65429 images of fruits spread across 95 labels. The data set is available on GitHub [18] and Kaggle [19]. The labels and the number of images for training are given in Table 1.

Table 1: Number of images for each fruit. There are multiple varieties of apples each of them being considered as a separate object. We did not find the scientific/popular name for each apple so we labeled with digits (e.g. apple red 1, apple red 2 etc).

Label	Number of training images	Number of test images
Apple Braeburn	492	164
Apple Golden 1	492	164
Apple Golden 2	492	164
Apple Golden 3	481	161
Apple Granny Smith	492	164
Apple Red 1	492	164
Apple Red 2	492	164
Apple Red 3	429	144
Apple Red Delicious	490	166
Apple Red Yellow 1	492	164
Apple Red Yellow 2	672	219
Apricot	492	164
Avocado	427	143
Avocado ripe	491	166
Banana	490	166
Banana Lady Finger	450	152
Banana Red	490	166
Cactus fruit	490	166
Cantaloupe 1	492	164
Continued on next page		

Table 1 – continued from previous page

Label	Number of training images	Number of test images
Cantaloupe 2	492	164
Carambula	490	166
Cherry 1	492	164
Cherry 2	738	246
Cherry Rainier	738	246
Cherry Wax Black	492	164
Cherry Wax Red	492	164
Cherry Wax Yellow	492	164
Chestnut	450	153
Clementine	490	166
Cocos	490	166
Dates	490	166
Granadilla	490	166
Grape Blue	984	328
Grape Pink	492	164
Grape White	490	166
Grape White 2	490	166
Grape White 3	492	164
Grape White 4	471	158
Grapefruit Pink	490	166
Grapefruit White	492	164
Guava	490	166
Hazelnut	464	157
Huckleberry	490	166
Kaki	490	166
Kiwi	466	156
Kumquats	490	166
Lemon	492	164
Lemon Meyer	490	166
Limes	490	166
Lychee	490	166
Mandarine	490	166
Mango	490	166
Mangostan	300	102

Table 1 – continued from previous page

Label	Number of training images	Number of test images
Maracuja	490	166
Melon Piel de Sapo	738	246
Mulberry	492	164
Nectarine	492	164
Orange	479	160
Papaya	492	164
Passion Fruit	490	166
Peach	492	164
Peach 2	738	246
Peach Flat	492	164
Pear	492	164
Pear Abate	490	166
Pear Kaiser	300	102
Pear Monster	490	166
Pear Williams	490	166
Pepino	490	166
Physalis	492	164
Physalis with Husk	492	164
Pineapple	490	166
Pineapple Mini	493	163
Pitahaya Red	490	166
Plum	447	151
Plum 2	420	142
Plum 3	900	304
Pomegranate	492	164
Pomelo Sweetie	450	153
Quince	490	166
Rambutan	492	164
Raspberry	490	166
Redcurrant	492	164
Salak	490	162
Strawberry	492	164
Strawberry Wedge	738	246
Tamarillo	490	166
		Continued on next page

Table 1 – continued from previous page

Label	Number of training images	Number of test images
Tangelo	490	166
Tomato 1	738	246
Tomato 2	672	225
Tomato 3	738	246
Tomato 4	479	160
Tomato Cherry Red	492	164
Tomato Maroon	367	127
Walnut	735	249

5 TensorFlow library

For the purpose of implementing, training and testing the network described in this paper we used the TensorFlow library [32]. This is an open source framework for machine learning created by Google for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays called tensors.

The main components in a TensorFlow system are the client, which uses the Session interface to communicate with the master, and one or more worker processes, with each worker process responsible for arbitrating access to one or more computational devices (such as CPU cores or GPU cards) and for executing graph nodes on those devices as instructed by the master.

TensorFlow offers some powerful features such as: it allows computation mapping to multiple machines, unlike most other similar frameworks; it has built in support for automatic gradient computation; it can partially execute subgraphs of the entire graph and it can add constraints to devices, like placing nodes on devices of a certain type, ensure that two or more objects are placed in the same space etc.

TensorFlow is used in several projects, such as the Inception Image Classification Model [31]. This project introduced a state of the art network for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014. In this project the usage of the computing resources is improved by adjusting the network width and depth while keeping the computational budget constant[31].

Another project that employs the TensorFlow framework is DeepSpeech, developed by Mozilla. It is an open source Speech-To-Text engine based on Baidu's Deep Speech architecture [9]. The architecture is a state of the art recognition system developed using end-to-end deep learning. It is simpler that other architectures and does not need hand designed components for background noise, reverberation or speaker variation.

We will present the most important utilized methods and data types from TensorFlow together with a short description for each of them.

A convolutional layer is defined like this:

```
1      conv2d(
2          input,
3          filter,
4          strides,
5          padding,
6          use_cudnn_on_gpu=True,
7          data_format='NHWC',
8          dilations=[1, 1, 1, 1],
9          name=None
10     )
```

Computes a 2-D convolution given 4-D *input* and *filter* tensors. Given an input tensor of shape [*batch, in_height, in_width, in_channels*] and a kernel tensor of shape [*filter_height, filter_width, in_channels, out_channels*], this op performs the following:

- Flattens the filter to a 2-D matrix with shape [*filter_height * filter_width * in_channels, output_channels*].

- Extracts image patches from the input tensor to form a virtual tensor of shape [*batch, out_height, out_width, filter_height * filter_width * in_channels*].

- For each patch, right-multiplies the filter matrix and the image patch vector.

```
1      tf.nn.max_pool(
2          value,
3          ksize,
4          strides,
5          padding,
6          data_format='NHWC',
7          name=None
8      )
```

Performs the max pooling operation on the input. The *ksize* and *strides* parameters can be tuples or lists of tuples of 4 elements. *Ksize* represents the size of the window for each dimension of the input tensor and *strides* represents the stride of the sliding window for each dimension of the input tensor. The *padding* parameter can be ''VALID'' or ''SAME''.

```
1          tf.nn.relu(
2              features,
3              name=None
4          )
```

Computes the rectified linear operation - max(features, 0). *Features* is a tensor.

```
1          tf.nn.dropout(
2              x,
3              keep_prob,
4              noise_shape=None,
5              seed=None,
6              name=None
7          )
```

Applies dropout on input *x* with probability *keep_prob*. This means that for each value in *x* the method outputs the value scaled by 1 / *keep_prob* with probability *keep_prob* or 0. The scaling is done on order to preserve the sum of the elements. The *noise_shape* parameter defines which groups of values are kept or dropped together. For example, a value of [*k, 1, 1, n*] for the *noise_shape*, with *x* having the shape [*k, l, m, n*], means that each row and column will be kept or dropped together, while the batch and channel components will be kept or dropped separately.

6 The structure of the neural network used in experiments

For this project we used a convolutional neural network. As previously described this type of network makes use of convolutional layers, pooling layers, ReLU layers, fully connected layers and loss layers. In a typical CNN architecture, each convolutional layer is followed by a Rectified Linear Unit (ReLU) layer, then a Pooling layer then one or more convolutional layer and finally one or more fully connected layer.

Note again that a characteristic that sets apart the CNN from a regular neural network is taking into account the structure of the images while processing them. A regular neural network converts the input in a one dimensional array which makes the trained classifier less sensitive to positional changes.

The input that we used consists of standard RGB images of size 100 x 100 pixels.

The neural network that we used in this project has the structure given in Table 2.

Table 2: The structure of the neural network used in this paper.

Layer type	Dimensions	Output
Convolutional	5 x 5 x 4	16
Max pooling	2 x 2 — Stride: 2	-
Convolutional	5 x 5 x 16	32
Max pooling	2 x 2 — Stride: 2	-
Convolutional	5 x 5 x 32	64
Max pooling	2 x 2 — Stride: 2	-
Convolutional	5 x 5 x 64	128
Max pooling	2 x 2 — Stride: 2	-
Fully connected	5 x 5 x 128	1024
Fully connected	1024	256
Softmax	256	60

A visual representation of the neural network used is given in Figure 2.

- The first layer (Convolution #1) is a convolutional layer which applies 16 5 x 5 filters. On this layer we apply max pooling with a filter of shape 2 x 2 with stride 2 which specifies that the pooled regions do not overlap (Max-Pool #1). This also reduces the width and height to 50 pixels each.

- The second convolutional (Convolution #2) layer applies 32 5 x 5 filters which outputs 32 activation maps. We apply on this layer the same kind of max pooling(Max-Pool #2) as on the first layer, shape 2 x 2 and stride 2.

- The third convolutional (Convolution #3) layer applies 64 5 x 5 filters. Following is another max pool layer(Max-Pool #3) of shape 2 x 2 and stride 2.

- The fourth convolutional (Convolution #4) layer applies 128 5 x 5 filters after which we apply a final max pool layer (Max-Pool #4).
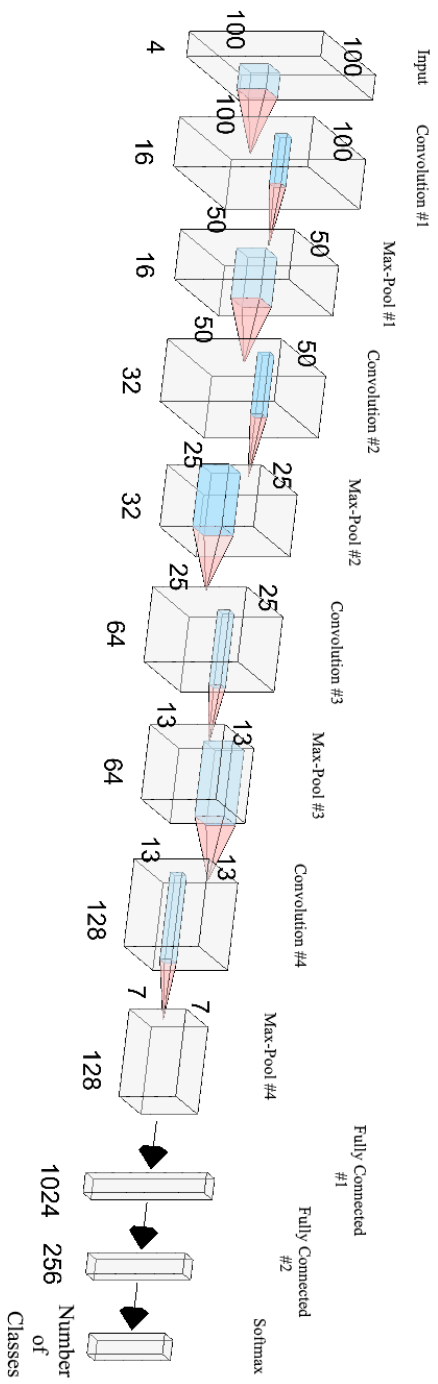
Figure 2: Graphical representation of the convolutional neural network used in experiments.

- Because of the four max pooling layers, the dimensions of the representation have each been reduced by a factor of 16, therefore the fifth layer, which is a fully connected layer(Fully Connected #1), has 7 x 7 x 16 inputs.

- This layer feeds into another fully connected layer (Fully Connected #2) with 1024 inputs and 256 outputs.

- The last layer is a softmax loss layer (Softmax) with 256 inputs. The number of outputs is equal to the number of classes.

We present a short scheme containing the flow of the the training process:

```
iterations = 75000

read_images(images)
apply_random_hue_saturation_changes(images)
apply_random_vertical_horizontal_flips(images)
convert_to_hsv(images)
add_grayscale_layer(images)

define_network_structure(images, network,
        training_operation)

for i in range(1, iterations):
            sess.run(training_operation)
```

7   Numerical experiments

The dataset was split in 2 parts: training set - which consists of 48905 images of fruits and testing set - which is made of 16421 images.

The data was bundled into a TFRecords file (specific to TensorFlow). This is a binary file that contains protocol buffers with a feature map. In this map it is possible to store information such as the image height, width, depth and even the raw image. Using these files we can create queues in order to feed the data to the neural network.

By calling the method *shuffle_batch* we provide randomized input to the network. The way we used this method was providing it example tensors for images and labels and it returned tensors of shape batch size x image

dimensions and batch size x labels. This helps greatly lower the chance of using the same batch multiple times for training, which in turn improves the quality of the network.

We ran multiple scenarios in which the neural network was trained using different levels of data augmentation and preprocessing:

- convert the input RGB images to grayscale

- keep the input images in the RGB colorspace

- convert the input RGB images to the HSV colorspace

- convert the input RGB images to the HSV colorspace and to grayscale and merge them

- apply random hue and saturation changes on the input RGB images, randomly flip them horizontally and vertically, then convert them to the HSV colorspace and to grayscale and merge them

For each scenario we used the previously described neural network which was trained over 75000 iterations with batches of 60 images selected at random from the training set. Every 50 steps we calculated the accuracy using cross-validation. For testing we ran the trained network on the test set. The results for each case are presented in Table 3.

Table 3: Results of training the neural network on the fruits-360 dataset.

Scenario	Accuracy on training set	Accuracy on test set
Grayscale	99.96%	94.24%
RGB	99.23%	93.47%
HSV	99.98%	97.01%
HSV + Grayscale	99.78%	95.71%
HSV + Grayscale + hue/saturation change + flips	99.86%	97.04%

As reflected in Table 3 the best results were obtained by applying data augmentation and converting the RGB images to the HSV colorspace to which the grayscale representation was added. This is intuitive since in this scenario we attach the most amount of information to the input, thus

the network can learn multiple features in order to classify the images.

It is also important to notice that training the grayscale images only yielded the best results on the train set but very weak results on the test set. We investigated this problem and we have discovered that a lot of images containing apples are incorrectly classified on the test set. In order to further investigate the issue we ran a round of training and testing on just the apple classes of images. The results were similar, with high accuracy on the train data, but low accuracy on the test data. We attribute this to overfitting, because the grayscale images lose too many features, the network does not learn properly how to classify the images.

In order to determine the best network configuration for classifying the images in out dataset, we took multiple configurations, used the train set to train them and then calculated their accuracy on the test and training set. In Table 4 we present the results.

Table 4: Results of training different network configurations on the fruits-360 dataset.

Nr.	Configuration			Accuracy on training set	Accuracy on test set
1	Convolutional	5 x 5	16	99.62%	95.88%
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	128		
	Fully connected	-	1024		
	Fully connected	-	256		
2	Convolutional	5 x 5	8	99.14%	95.85%
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	128		
	Fully connected	-	1024		
	Fully connected	-	256		
3	Convolutional	5 x 5	32	99.59%	95.27%
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	128		
	Fully connected	-	1024		
	Fully connected	-	256		

Table 4: Results of training different network configurations on the fruits-360 dataset.

Nr.	Configuration			Accuracy on training set	Accuracy on test set
4	Convolutional	5 x 5	16	99.04%	93.99%
	Convolutional	5 x 5	16		
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	128		
	Fully connected	-	1024		
	Fully connected	-	256		
5	Convolutional	5 x 5	16	99.22%	94.70%
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	128		
	Fully connected	-	1024		
	Fully connected	-	256		
6	Convolutional	5 x 5	16	98.40%	92.91%
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	128		
	Fully connected	-	1024		
	Fully connected	-	256		
7	Convolutional	5 x 5	16	99.55%	94.86%
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	128		
	Convolutional	5 x 5	128		
	Fully connected	-	1024		
	Fully connected	-	256		
8	Convolutional	5 x 5	16	99.49%	95.09%
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	64		
	Fully connected	-	1024		
	Fully connected	-	256		

Table 4: Results of training different network configurations on the fruits-360 dataset.

Nr.	Configuration			Accuracy on training set	Accuracy on test set
9	Convolutional	5 x 5	16	99.22%	95.23%
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	128		
	Fully connected	-	512		
	Fully connected	-	256		
10	Convolutional	5 x 5	16	99.51%	94.64%
	Convolutional	5 x 5	32		
	Convolutional	5 x 5	64		
	Convolutional	5 x 5	128		
	Fully connected	-	1024		
	Fully connected	-	512		

From Table 4 we can see that the best performance on the test set was obtained by configuration nr. 1. The same configuration obtained the best accuracy on the training set. The general trend indicates that if a configuration obtained high accuracy on the train set, this will translate into a good performance on the test set. However, a few outliers can be seen: configuration nr. 10 obtained 99.51% accuracy on the train set, but only 94.64% accuracy on the test set, being the third worst. This is a result of the model overfitting to the training data and not properly generalizing to other images. The reverse can be seen with configuration nr. 2, which obtained the third worst performance on training data, but the second best accuracy on the test set.

The evolution of accuracy during training is given in Figure 3. It can be seen that the training rapidly improves in the first 1000 iterations (accuracy becomes greater than 90%) and then it is very slowly improved in the next 74000 iterations.

Some of the incorrectly classified images are given in Table 5.

Table 5: Some of the images that were classified incorrectly. On the top we have the correct class of the fruit and on the bottom we have the class (and its associated probability) that was assigned by the network.

Apple Golden 2	Apple Golden 3	Braeburn(Apple)	Peach
Apple Golden 3	Granny Smith (Apple)	Apple Red 2	Apple Red Yellow
96.54%	95.22%	97.71%	97.85%
Pomegranate	Peach	Pear	Pomegranate
Nectarine	Apple Red 1	Apple Golden 2	Braeburn(Apple)
94.64%	97.87%	98.73%	97.21%

8   Conclusions and further work

We described a new and complex database of images with fruits. Also we made some numerical experiments by using TensorFlow library in order to classify the images according to their content.

From our point of view one of the main objectives for the future is to improve the accuracy of the neural network. This involves further experimenting with the structure of the network. Various tweaks and changes to
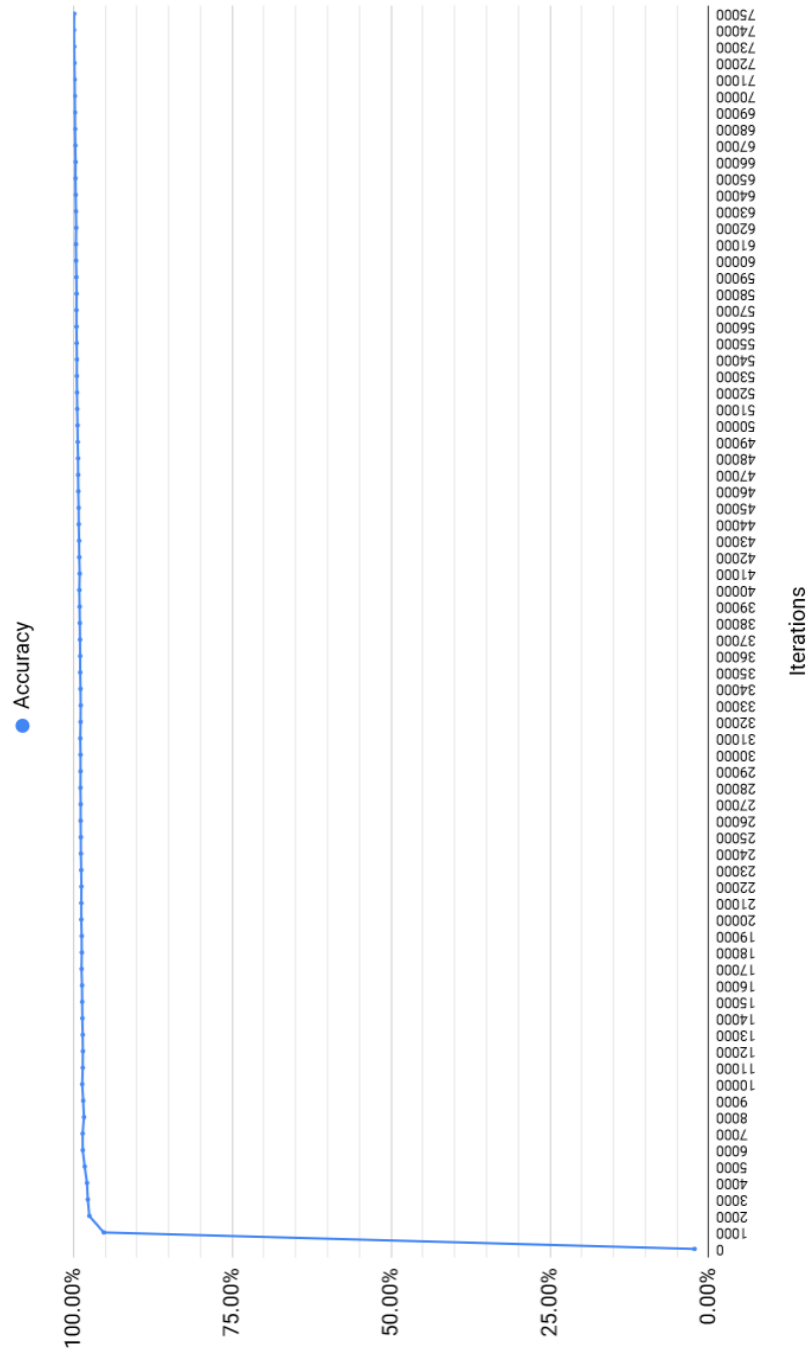
Figure 3: Accuracy evolution over 75000 training iterations

any layers as well as the introduction of new layers can provide completely different results. Another option is to replace all layers with convolutional layers. This has been shown to provide some improvement over the networks that have fully connected layers in their structure. A consequence of replacing all layers with convolutional ones is that there will be an increase in the number of parameters for the network [29]. Another possibility is to replace the rectified linear units with exponential linear units. According to paper [8], this reduces computational complexity and add significantly better generalization performance than rectified linear units on networks with more that 5 layers. We would like to try out these practices and also to try to find new configurations that provide interesting results.

In the near future we plan to create a mobile application which takes pictures of fruits and labels them accordingly.

Another objective is to expand the data set to include more fruits. This is a more time consuming process since we want to include items that were not used in most others related papers.

Acknowledgments

Appendix

In this section we present the source code and project structure used in the numerical experiment described in this paper. The source code can be downloaded from GitHub [18].

The source code is organized (on GitHub [18]) as follows:

```
root_directory
    fruit_detection
        detect_fruits.py
    network
        fruit_test_net.py
        fruit_train_net.py
    network_structure
        fruit_network.py
        utils.py
    utils
        build_image_data.py
        constants.py
        freeze_graph.py
        labels
```

In order to run the project from the command line, first make sure the PYTHONPATH system variable contains the path to the root_directory. Ensure that the **utils/constants.py** contains the proper paths.

Run the **utils/build_image_data.py** to generate the tfrecord files with training and test data. This script is provided in the tensorFlow library. The file contains several default values for the flags. They can be changed in the code directly or different values can be provided from the command line:
python utils/build_image_data.py [flags]

where flags can be:

- –train_directory=path to the folder containing the train images

- –validation_directory=path to the folder containing the validation images

- –output_directory=path to where to output the tfrecord files

- –labels_file=path to the label file

- –train_shards, –test_shards determine the number of tfrecord files for train data and test data

- –num_threads is the number of threads to create when creating the tfrecord files

After the train and test data has been serialized, the train and test scripts can be run:
python network/fruit_train_net.py
python network/fruit_test_net.py
After the training has completed, the python utils/build_image_data.py [flags] script can be run:
python utils/build_image_data.py –image_path="path to a jpeg file"
Finally, the **utils/freeze_graph.py** script, which is also provided as a utility script in tensorFlow, creates a single file with the trained model data.
python freeze_graph flags
These flags are mandatory:

- –input_graph=path to the pbtxt file

- –input_checkpoint=path to the ckpt file

- –output_graph=name of the output file

- –output_node_names=name of the last layer of the network (found in the network_structure/fruit_network.py file, in the conv_net method, in this case the name of the last layer is "out/out")

In the following, we will provide explanations for the code. We will begin with the definition of the general parameters and configurations of the project.
The following are defined in the **utils/constants.py** file:

- *root_dir* - the top level folder of the project

- *data_dir* - the folder where the .tfrecords are persisted

- *fruit_models_dir* - the folder where the network structure and parameters are saved

- *labels_file* - the path to the file that contains all the labels used

- *training_images_dir, test_images_dir* - paths to the folders containing the training and test images

- *num_classes* - the number of different classes used

– it is determined by counting the number of elements in the labels file, which is also used in the **utils/freeze_graph.py** script

- *number_train_images, number_test_images* - number of training and test images; used in the test method to calculate accuracy

All these configurations can be changed to suit the setup of anyone using the code.

```
1  utils/constants.py
2  import os
3
4  # needs to be changed according to the location of the
       project
5  root_dir = 'C:\\root_directory\\'
6  data_dir = root_dir + '\\data\\'
7  fruit_models_dir = root_dir + '\\fruit_models\\'
8  labels_file = root_dir + '\\utils\\labels'
9
10 # change this to the path of the folders that hold the
       images
11 training_images_dir = '\\Fruit-Images-Dataset\\
       Training'
12 test_images_dir = '\\Fruit-Images-Dataset\\Test'
13
14 # number of classes: number of fruit classes + 1
       resulted due to the build_image_data.py script that
       leaves the first class as a background class
15 # using the labels file that is also used in the
       build_image_data.py
16 with open(labels_file) as f:
17     labels = f.readlines()
18     num_classes = len(labels) + 1
19 number_train_images = 48905
20 number_test_images = 16421
```

In the **network_structure/utils.py** file we have helper methods used across the project:

- *conv, fully_connected* combine the TensorFlow methods of defining a convolutional layer and a fully connected layer, respectively, adding the bias to the layer and applying a linear rectifier

- – a convolutional layer consists of groups of neurons that make up kernels
- – the kernels have a small size but they always have the same depth as the input
- – the neurons from a kernel are connected to a small region of the input, called the receptive field, because it is highly inefficient to link all neurons to all previous outputs in the case of inputs of high dimensions such as images

- *max_pool, loss, _int64_feature and _bytes_feature* simplify the calls to the corresponding TensorFlow methods

- *parse_single_example* converts a serialized input into an image and label as they were saved using the *utils/build_image_data.py*

Here we also define methods to perform data augmentation on the input images. Data augmentation is a good way to reduce overfitting on models. Flipping the image horizontally and vertically helps prevent the use the orientation of the fruit as a feature when training. This should result in fruits being correctly classified regardless of their position in an image.

- *augment_image* applies the following operations on the train images

  1. Alters the hue of the image
  2. Alters the saturation of the image
  3. Flips the image horizontally
  4. Flips the image vertically
  5. Calls the *build_hsv_grayscale_image* on the result

- *build_hsv_grayscale_image* converts the image to HSV color space and creates a grayscale version of the image and adds it as a fourth channel to the HSV image By altering the hue and saturation, we simulate having a larger variety of fruits in the images. The values with which we alter these properties are small, since in nature there is a small color variance between different fruits of the same species

```
1 network_structure/utils.py
2
```

```python
import tensorflow as tf


# perform data augmentation on images
# add random hue and saturation
# randomly flip the image vertically and horizontally
# converts the image from RGB to HSV and
# adds a 4th channel to the HSV ones that contains the
    image in gray scale
def augment_image(image):
    image = tf.image.convert_image_dtype(image, tf.
        float32)
    image = tf.image.random_hue(image, 0.02)
    image = tf.image.random_saturation(image, 0.9,
        1.2)
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_flip_up_down(image)
    return build_hsv_grayscale_image(image)


# convert the image to HSV and add the gray scale
    channel
def build_hsv_grayscale_image(image):
    image = tf.image.convert_image_dtype(image, tf.
        float32)
    gray_image = tf.image.rgb_to_grayscale(image)
    image = tf.image.rgb_to_hsv(image)
    rez = tf.concat([image, gray_image], 2)
    return rez


def parse_single_example(serialized_example):
    features = tf.parse_single_example(
        serialized_example,
        features={
            'image_raw': tf.FixedLenFeature([], tf.
                string),
            'label': tf.FixedLenFeature([], tf.int64),
```

```
35          'height': tf.FixedLenFeature([], tf.int64)
                ,
36          'width': tf.FixedLenFeature([], tf.int64)
37        }
38      )
39      image = tf.image.decode_jpeg(features['image_raw'
          ], channels=3)
40      image = tf.reshape(image, [100, 100, 3])
41      label = tf.cast(features['label'], tf.int32)
42      return image, label
43
44
45 def conv(input_tensor, name, kernel_width,
      kernel_height, num_out_activation_maps,
      stride_horizontal=1, stride_vertical=1,
      activation_fn=tf.nn.relu):
46      prev_layer_output = input_tensor.get_shape()[-1].
          value
47      with tf.variable_scope(name):
48          weights = tf.get_variable('weights', [
              kernel_height, kernel_width,
              prev_layer_output, num_out_activation_maps
              ], tf.float32,
49                                      tf.
                                        truncated_normal_initializer
                                        (stddev=5e-2,
                                        dtype=tf.float32)
                                        )
50          biases = tf.get_variable("bias", [
              num_out_activation_maps], tf.float32, tf.
              constant_initializer(0.0))
51          conv_layer = tf.nn.conv2d(input_tensor,
              weights, (1, stride_horizontal,
              stride_vertical, 1), padding='SAME')
52          activation = activation_fn(tf.nn.bias_add(
              conv_layer, biases), name=name)
53          return activation
54
```

```
55
56 def fully_connected(input_tensor, name, output_neurons
       , activation_fn=tf.nn.relu):
57     n_in = input_tensor.get_shape()[-1].value
58     with tf.variable_scope(name):
59         weights = tf.get_variable('weights', [n_in,
               output_neurons], tf.float32,
60                                   initializer=tf.
                                       truncated_normal_initializer
                                       (stddev=5e-2,
                                       dtype=tf.float32)
                                       )
61         biases = tf.get_variable("bias", [
               output_neurons], tf.float32, tf.
               constant_initializer(0.0))
62         logits = tf.nn.bias_add(tf.matmul(input_tensor
               , weights), biases, name=name)
63         if activation_fn is None:
64             return logits
65         return activation_fn(logits)
66
67
68 def max_pool(input_tensor, name, kernel_height,
       kernel_width, stride_horizontal, stride_vertical):
69     return tf.nn.max_pool(input_tensor,
70                           ksize=[1, kernel_height,
                                   kernel_width, 1],
71                           strides=[1,
                                   stride_horizontal,
                                   stride_vertical, 1],
72                           padding='VALID',
73                           name=name)
74
75
76 def loss(logits, onehot_labels):
77     xentropy = tf.nn.softmax_cross_entropy_with_logits
           (logits=logits, labels=onehot_labels, name='
           xentropy')
```

```
78    loss = tf.reduce_mean(xentropy, name='loss')
79    return loss
80
81
82 def _int64_feature(value):
83    if not isinstance(value, list):
84        value = [value]
85    return tf.train.Feature(int64_list=tf.train.
          Int64List(value=value))
86
87
88 def _bytes_feature(value):
89    return tf.train.Feature(bytes_list=tf.train.
          BytesList(value=[value]))
```

Following, in the **network_structure/fruit_network.py** file we have network parameters and the method that defines the network structure.

- *IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS* - the image height, width and depth respectively;

- *NETWORK_DEPTH* - the depth of the input for the network (3 from HSV image + 1 from grayscale image)

- *batch_size* - the number of images selected in each training/testing step

- *dropout* - the probability to keep a node in each training step

  - during training, at each iteration, some nodes are ignored with probability 1 − *dropout*
  - this results in a reduced network, which is then used for a forward or backward pass
  - dropout prevents neurons from developing co-dependency and, in turn, overfitting
  - outside of training, the dropout is ignored and the entire network is used for classifying

- *update_learning_rate* dynamically adjusts the learning rate as training progresses

- the weights and biases used are defined as follows:

- The first layer (Convolution #1) is a convolutional layer which applies 16 5 x 5 filters. On this layer we apply max pooling with a filter of shape 2 x 2 with stride 2 which specifies that the pooled regions do not overlap (Max-Pool #1). This also reduces the width and height to 50 pixels each.
- The second convolutional (Convolution #2) layer applies 32 5 x 5 filters which outputs 32 activation maps. We apply on this layer the same kind of max pooling(Max-Pool #2) as on the first layer, shape 2 x 2 and stride 2.
- The third convolutional (Convolution #3) layer applies 64 5 x 5 filters. Following is another max pool layer(Max-Pool #3) of shape 2 x 2 and stride 2.
- The fourth convolutional (Convolution #4) layer applies 128 5 x 5 filters after which we apply a final max pool layer (Max-Pool #4).
- Because of the four max pooling layers, the dimensions of the representation have each been reduced by a factor of 16, therefore the fifth layer, which is a fully connected layer(Fully Connected #1), has 7 x 7 x 16 inputs.
- This layer feeds into another fully connected layer (Fully Connected #2) with 1024 inputs and 256 outputs.
- The last layer is a softmax loss layer (Softmax) with 256 inputs. The number of outputs is equal to the number of classes.

- *build_model* define operations for the training process and for loss and accuracy evaluation

```
1 network_structure/fruit_network.py
2
3 import tensorflow as tf
4 import numpy as np
5 from . import utils
6 from utils import constants
7
8 HEIGHT = 100
9 WIDTH = 100
10 # number of channels for an image - jpeg image has RGB
       channels
```

```
11  CHANNELS = 3
12  # number of channels for the input layer of the
        network: HSV + gray scale
13  NETWORK_DEPTH = 4
14
15  batch_size = 60
16  input_size = HEIGHT * WIDTH * NETWORK_DEPTH
17  # probability to keep the values after a training
        iteration
18  dropout = 0.8
19
20  # placeholder for input layer
21  X = tf.placeholder(tf.float32, [None, input_size],
        name="X")
22  # placeholder for actual labels
23  Y = tf.placeholder(tf.int64, [None], name="Y")
24
25  initial_learning_rate = 0.001
26  final_learning_rate = 0.00001
27  learning_rate = initial_learning_rate
28
29
30  def conv_net(input_layer):
31      # number of activation maps for each convolutional
            layer
32      number_of_act_maps_conv1 = 16
33      number_of_act_maps_conv2 = 32
34      number_of_act_maps_conv3 = 64
35      number_of_act_maps_conv4 = 128
36
37      # number of outputs for each fully connected layer
38      number_of_fcl_outputs1 = 1024
39      number_of_fcl_outputs2 = 256
40
41      input_layer = tf.reshape(input_layer, shape=[-1,
            HEIGHT, WIDTH, NETWORK_DEPTH])
42
43      conv1 = utils.conv(input_layer, 'conv1',
```

```
        kernel_width=5, kernel_height=5,
        num_out_activation_maps=
        number_of_act_maps_conv1)
44    conv1 = utils.max_pool(conv1, 'max_pool1',
        kernel_height=2, kernel_width=2,
        stride_horizontal=2, stride_vertical=2)
45
46    conv2 = utils.conv(conv1, 'conv2', kernel_width=5,
          kernel_height=5, num_out_activation_maps=
        number_of_act_maps_conv2)
47    conv2 = utils.max_pool(conv2, 'max_pool2',
        kernel_height=2, kernel_width=2,
        stride_horizontal=2, stride_vertical=2)
48
49    conv3 = utils.conv(conv2, 'conv3', kernel_width=5,
          kernel_height=5, num_out_activation_maps=
        number_of_act_maps_conv3)
50    conv3 = utils.max_pool(conv3, 'max_pool3',
        kernel_height=2, kernel_width=2,
        stride_horizontal=2, stride_vertical=2)
51
52    conv4 = utils.conv(conv3, 'conv4', kernel_width=5,
          kernel_height=5, num_out_activation_maps=
        number_of_act_maps_conv4)
53    conv4 = utils.max_pool(conv4, 'max_pool4',
        kernel_height=2, kernel_width=2,
        stride_horizontal=2, stride_vertical=2)
54
55    flattened_shape = np.prod([s.value for s in conv4.
        get_shape()[1:]])
56    net = tf.reshape(conv4, [-1, flattened_shape],
        name="flatten")
57
58    fcl1 = utils.fully_connected(net, 'fcl1',
        number_of_fcl_outputs1)
59    fcl1 = tf.nn.dropout(fcl1, dropout)
60
61    fcl2 = utils.fully_connected(fcl1, 'fcl2',
```

```
           number_of_fcl_outputs2)
62        fcl2 = tf.nn.dropout(fcl2, dropout)

63

64        out = utils.fully_connected(fcl2, 'out', constants
              .num_classes, activation_fn=None)

65

66        return out

67

68

69  def update_learning_rate(acc, learn_rate):
70        return max(learn_rate - acc * learn_rate * 0.9,
              final_learning_rate)

71

72

73  def build_model():
74        # build the network
75        logits = conv_net(input_layer=X)
76        # apply softmax on the final layer
77        prediction = tf.nn.softmax(logits)

78

79        # calculate the loss using the predicted labels vs
               the expected labels
80        loss = tf.reduce_mean(tf.nn.
              sparse_softmax_cross_entropy_with_logits(logits
              =logits, labels=Y))
81        # use adaptive moment estimation optimizer
82        optimizer = tf.train.AdamOptimizer(learning_rate=
              learning_rate)
83        train_op = optimizer.minimize(loss=loss)

84

85        # calculate the accuracy for this training step
86        correct_prediction = tf.equal(tf.argmax(prediction
              , 1), Y)

87

88        return train_op, loss, correct_prediction
```

The following 2 files, **network/fruit_test_net.py, network/fruit_train_net.py**
contain the logic for training and testing the network Firstly, in **network/fruit_train_net.py**
we have:

- *iterations* - the number of steps for which the training will be done

- *acc_display_interval* - the number of iterations to train for before displaying the loss and accuracy of the network

- *save_interval* - default number of iterations after we save the model

- *step_display_interval* - number of iterations after we display the total number of steps done and the time spent training the past step_display_interval iterations

- *useCkpt* - if true, load a previously trained model and continue training, else, train a new model from scratch

- *build_datasets* - read the tfrecord files and prepare two datasets to be used during the training process

- *calculate_intermediate_accuracy_and_loss* - calculates the loss and accuracy on the training dataset; used during training to monitor the performance of the network

- *train_model* - runs the training process

```
1 network/fruit_train_net.py
2
3 import tensorflow as tf
4 import numpy as np
5 import time
6 import os
7 import re
8
9 from network_structure import fruit_network as network
10 from network_structure import utils
11
12 from utils import constants
13
14 # default number of iterations to run the training
15 iterations = 75000
16 # default number of iterations after we display the
       loss and accuracy
17 acc_display_interval = 1000
```

```python
18 # default number of iterations after we save the model
19 save_interval = 1000
20 # default number of iterations after we display the
       total number of steps done and the time spent
       training the past step_display_interval iterations
21 step_display_interval = 100
22 # use the saved model and continue training; defaults
       to false
23 useCkpt = False
24
25 # create two datasets from the previously created
       training tfrecord files
26 # the first dataset will apply data augmentation and
       shuffle its elements and will continuously queue
       new items - used for training
27 # the second dataset will iterate once over the
       training images - used for evaluating the loss and
       accuracy during the training
28 def build_datasets(filenames, batch_size):
29     train_dataset = tf.data.TFRecordDataset(filenames)
           .repeat()
30     train_dataset = train_dataset.map(utils.
           parse_single_example).map(lambda image, label:
           (utils.augment_image(image), label))
31     train_dataset = train_dataset.shuffle(buffer_size
           =10000, reshuffle_each_iteration=True)
32     train_dataset = train_dataset.batch(batch_size)
33     test_dataset = tf.data.TFRecordDataset(filenames)
34     test_dataset = test_dataset.map(utils.
           parse_single_example).map(lambda image, label:
           (utils.build_hsv_grayscale_image(image), label)
           )
35     test_dataset = test_dataset.batch(batch_size)
36     return train_dataset, test_dataset
37
38
39 def train_model(session, train_operation,
       loss_operation, correct_prediction, iterator_map):
```

```
40    time1 = time.time()
41    train_iterator = iterator_map["train_iterator"]
42    test_iterator = iterator_map["test_iterator"]
43    test_init_op = iterator_map["test_init_op"]
44    train_images_with_labels = train_iterator.get_next
          ()
45    test_images_with_labels = test_iterator.get_next()
46    for i in range(1, iterations + 1):
47        batch_x, batch_y = session.run(
              train_images_with_labels)
48        batch_x = np.reshape(batch_x, [network.
              batch_size, network.input_size])
49        session.run(train_operation, feed_dict={
              network.X: batch_x, network.Y: batch_y})
50
51        if i % step_display_interval == 0:
52            time2 = time.time()
53            print("time: %.4f step: %d" % (time2 -
                  time1, i))
54            time1 = time.time()
55
56        if i % acc_display_interval == 0:
57            acc_value, loss =
                  calculate_intermediate_accuracy_and_loss
                  (session, correct_prediction,
                  loss_operation, test_images_with_labels
                  , test_init_op, constants.
                  number_train_images)
58            network.learning_rate = network.
                  update_learning_rate(acc_value,
                  learn_rate=network.learning_rate)
59            print("step: %d loss: %.4f accuracy: %.4f"
                   % (i, loss, acc_value))
60        if i % save_interval == 0:
61            # save the weights and the meta data for
                  the graph
62            saver.save(session, constants.
                  fruit_models_dir + 'model.ckpt')
```

```python
63              tf.train.write_graph(session.graph_def,
                    constants.fruit_models_dir, 'graph.
                    pbtxt')
64
65
66  def calculate_intermediate_accuracy_and_loss(session,
        correct_prediction, loss_operation,
        test_images_with_labels, test_init_op,
        total_image_count):
67      sess.run(test_init_op)
68      loss = 0
69      predicted = 0
70      count = 0
71      while True:
72          try:
73              test_batch_x, test_batch_y = session.run(
                    test_images_with_labels)
74              test_batch_x = np.reshape(test_batch_x,
                    [-1, network.input_size])
75              l, p = session.run([loss_operation,
                    correct_prediction], feed_dict={network
                    .X: test_batch_x, network.Y:
                    test_batch_y})
76              loss += l
77              predicted += np.sum(p)
78              count += 1
79          except tf.errors.OutOfRangeError:
80              break
81      return predicted / total_image_count, loss / count
82
83
84  if __name__ == '__main__':
85
86      with tf.Session().as_default() as sess:
87
88          # input tfrecord files
89          tfrecords_files = [(constants.data_dir + f)
                for f in os.listdir(constants.data_dir) if
```

```
              re.match('train', f)]
90    train_dataset, test_dataset = build_datasets(
              filenames=tfrecords_files, batch_size=
              network.batch_size)
91    train_iterator = train_dataset.
              make_one_shot_iterator()
92    test_iterator = tf.data.Iterator.
              from_structure(test_dataset.output_types,
              test_dataset.output_shapes)
93    test_init_op = test_iterator.make_initializer(
              test_dataset)
94    iterator_map = {"train_iterator":
              train_iterator, "test_iterator":
              test_iterator, "test_init_op": test_init_op
              }
95
96    train_op, loss, correct_prediction = network.
              build_model()
97    init = tf.global_variables_initializer()
98    saver = tf.train.Saver()
99    sess.run(init)
100
101   # restore the previously saved value if we
              wish to continue the training
102   if useCkpt:
103       ckpt = tf.train.get_checkpoint_state(
              constants.fruit_models_dir)
104       saver.restore(sess, ckpt.
              model_checkpoint_path)
105
106   train_model(sess, train_op, loss,
              correct_prediction, iterator_map)
107
108   sess.close()
```

Secondly, in **network/fruit_test_net.py** we have:

- *useTrain* - if true, calculate overall accuracy on the train set, else, calculate accuracy over the test set

- *mislabeled* - map of pairs where the key is a string representing the label and an integer representing how many images from that class were incorrectly classified

- *build_dataset* - create datasets on the tfrecord files; one to retrieve data for training, one to show the evolution of the accuracy during training, one to evaluate the accuracy on the test set after the training

- *setup_test_data* - builds the *mislabeled* map and an array of human readable labels for the classes

- *test_model* - runs the testing process

```python
1  network/fruit_test_net.py
2
3  import numpy
4  import tensorflow as tf
5  import numpy as np
6  import os
7  import re
8
9  from network_structure import fruit_network as network
10 from network_structure import utils
11 from utils import constants
12
13
14 def setup_test_data():
15     # create a map to add for each label the number of
           images that were labeled incorrectly
16     mislabeled = {}
17
18     # associate the label number with the actual human
           readable label name
19     with open(constants.root_dir + '\\utils\\labels')
         as f:
20         labels_text = f.readlines()
21     labels_text = [x.strip() for x in labels_text]
22     for label in labels_text:
23         mislabeled[label] = 0
24
```

```python
25      # class 0 is background class so it's labeled as
            nothing
26      labels_text = ["nothing"] + labels_text
27
28      return mislabeled, labels_text
29
30
31  def build_dataset(filenames, batch_size):
32      test_dataset = tf.data.TFRecordDataset(filenames)
33      test_dataset = test_dataset.map(utils.
            parse_single_example).map(lambda image, label:
            (utils.build_hsv_grayscale_image(image), label)
            )
34      test_dataset = test_dataset.batch(batch_size)
35      return test_dataset
36
37
38  def test_model(sess, pred, iterator, total_images,
        file_name):
39      images_left_to_process = total_images
40      total_number_of_images = total_images
41      images_with_labels = iterator.get_next()
42      correct = 0
43      while True:
44          try:
45              batch_x, batch_y = sess.run(
                    images_with_labels)
46              batch_x = np.reshape(batch_x, [-1, network
                    .input_size])
47              # the results of the classification is an
                    array of 1 and 0, 1 is a correct
                    classification
48              results = sess.run(pred, feed_dict={
                    network.X: batch_x, network.Y: batch_y
                    })
49              images_left_to_process =
                    images_left_to_process - len(results)
50              for i in range(len(results)):
```

```python
                    if not results[i]:
                        mislabeled[labels_text[batch_y[i
                            ]]] += 1

                correct = correct + numpy.sum(results)
                print("Predicted %d out of %d; partial
                    accuracy %.4f" % (correct,
                    total_number_of_images -
                    images_left_to_process, correct / (
                    total_number_of_images -
                    images_left_to_process)))
        except tf.errors.OutOfRangeError:
            break
    print("Final accuracy on %s data: %.8f" % (
        file_name, correct / total_number_of_images))


if __name__ == '__main__':

    with tf.Session() as sess:
        useTrain = False
        if useTrain:
            file_name = 'train'
            total_images = constants.
                number_train_images
        else:
            file_name = 'test'
            total_images = constants.
                number_test_images

        mislabeled, labels_text = setup_test_data()
        _, _, prediction = network.build_model()
        init = tf.global_variables_initializer()

        saver = tf.train.Saver()
        sess.run(init)
        tfrecords_files = [(constants.data_dir + f)
            for f in os.listdir(constants.data_dir) if
```

```
           re.match(file_name, f)]
79         dataset = build_dataset(tfrecords_files,
               network.batch_size)
80         iterator = dataset.make_one_shot_iterator()
81         ckpt = tf.train.get_checkpoint_state(constants
               .fruit_models_dir)
82         saver.restore(sess, ckpt.model_checkpoint_path
               )
83         test_model(sess, prediction, iterator,
               total_images, file_name)
84         print(mislabeled)
85
86         sess.close()
```

Finally, we have the **fruit_detection/detect_fruits.py**. This serves as a basic example on how to read an image from a file, resize it and feed it through a trained model. The script prints the class that had the highest probability after passing it through the model.

```
1  fruit_detection/detect_fruits.py
2
3  import tensorflow as tf
4  from utils import constants
5
6  with open(constants.root_dir + '\\utils\\labels') as f
       :
7      labels = f.readlines()
8  labels = [x.strip() for x in labels]
9  labels = ["nothing"] + labels
10
11 tf.app.flags.DEFINE_string('image_path', 'images\\
       Lemon2.jpg', 'Path to image')
12 FLAGS = tf.app.flags.FLAGS
13
14
15 # load image
16 def read_image(image_path, image_reader):
17     filename_queue = tf.train.string_input_producer([
           image_path])
18     _, image_file = image_reader.read(filename_queue)
```

```
19    local_image = tf.image.decode_jpeg(image_file)
20    local_image = tf.image.convert_image_dtype(
          local_image, tf.float32)
21    gray_image = tf.image.rgb_to_grayscale(local_image
          )
22    local_image = tf.image.rgb_to_hsv(local_image)
23    shape = tf.shape(local_image)
24    local_height = shape[0]
25    local_width = shape[1]
26    local_depth = shape[2]
27    local_image = tf.reshape(local_image, [
          local_height, local_width, local_depth])
28    final_image = tf.concat([local_image, gray_image],
           2)
29    return final_image, local_height, local_width,
          local_depth + 1
30
31
32 def predict(sess, X, softmax, images):
33    images = sess.run(images)
34    # the result of running this method is an array of
           probabilities, where each index in the array
          corresponds to a label
35    probability = sess.run(softmax, feed_dict={X:
          images})
36    # get the highest probability from the array and
          that should be the result
37    prediction = sess.run(tf.argmax(probability, 1))
38    return prediction, probability[0][prediction]
39
40
41 def process_image(sess, X, softmax, image,
      image_height, image_width, image_depth):
42    image_depth = sess.run(image_depth)
43    image_height = sess.run(image_height)
44    image_width = sess.run(image_width)
45    # resize the image to 100 x 100 pixels and shape
          it to be like an array of one image, since that
```

```
            is the required input for the network
46      # for smaller parts of an image and feed those to
            the network, tensorflow has a method called "
            extract_image_patches"
47      img = tf.image.resize_images(tf.reshape(image,
            [-1, image_height, image_width, image_depth]),
            [100, 100])
48      img = tf.reshape(img, [-1, 100 * 100 * 4])
49      rez, prob = predict(sess, X, softmax, img)
50      print('Label index: %d - Label: %s - Probability:
            %.4f' % (rez, labels[rez[0]], prob))
51
52
53 with tf.Session() as sess:
54      image_path = FLAGS.image_path
55      image_reader = tf.WholeFileReader()
56
57      # restore the trained model from the saved
            checkpoint; provide the path to the meta file
58      saver = tf.train.import_meta_graph(constants.
            fruit_models_dir + 'model.ckpt.meta')
59      # provide the path to the folder containing the
            checkpoints
60      saver.restore(sess, tf.train.latest_checkpoint(
            constants.fruit_models_dir))
61      graph = tf.get_default_graph()
62
63      # to obtain a tensor from the saved model, we must
             get it by name, which is why we name the
            tensors when we create them
64      # even if there is only one tensor with a name, in
             the meta and checkpoint files it is saved as
            an array, so we have to provide the index of
            the
65      # tensor that we want to get -> thus we call "
            get_tensor_by_name(tensor_name:0)"
66
67      # obtain the input tensor by name
```

```
68    X = graph.get_tensor_by_name('X:0')
69    # obtain the output layer by name and apply
          softmax on in in order to obtain an output of
          probabilities
70    softmax = tf.nn.softmax(graph.get_tensor_by_name('
          out/out:0'))
71
72    image, height, width, depth = read_image(
          image_path, image_reader)
73    coord = tf.train.Coordinator()
74    threads = tf.train.start_queue_runners(sess=sess,
          coord=coord)
75    process_image(sess, X, softmax, image, height,
          width, depth)
76
77    coord.request_stop()
78    coord.join(threads)
79    sess.close()
```

References

[1] Bargoti, S., and Underwood, J. Deep fruit detection in orchards. In *2017 IEEE International Conference on Robotics and Automation (ICRA)* (May 2017), pp. 3626–3633. ⇒ 4

[2] Barth, R., IJsselmuiden, J., Hemming, J., and Henten, E. V. Data synthesis methods for semantic segmentation in agriculture: A capsicum annuum dataset. *Computers and Electronics in Agriculture 144* (2018), 284 – 296. ⇒ 4

[3] Chan, T. F., and Vese, L. A. Active contours without edges. *IEEE Transactions on Image Processing 10*, 2 (Feb 2001), 266–277. ⇒ 5

[4] Cheng, H., Damerow, L., Sun, Y., and Blanke, M. Early yield prediction using image analysis of apple fruit and tree canopy features with neural networks. *Journal of Imaging 3*, 1 (2017). ⇒ 4

[5] Cireşan, D. C., Giusti, A., Gambardella, L. M., and Schmidhuber, J. Deep neural networks segment neuronal membranes in electron microscopy images. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2* (USA, 2012), NIPS'12, Curran Associates Inc., pp. 2843–2851. ⇒ 5

[6] Cireşan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Schmidhuber, J. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two* (2011), IJCAI'11, AAAI Press, pp. 1237–1242. ⇒ 5

[7] Ciresan, D. C., Meier, U., and Schmidhuber, J. Multi-column deep neural networks for image classification. *CoRR abs/1202.2745* (2012). ⇒ 6

[8] Clevert, D., Unterthiner, T., and Hochreiter, S. Fast and accurate deep network learning by exponential linear units (elus). *CoRR abs/1511.07289* (2015). ⇒ 26

[9] Hannun, A. Y., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., and Ng, A. Y. Deep speech: Scaling up end-to-end speech recognition. *CoRR abs/1412.5567* (2014). ⇒ 14

[10] Hemming, J., Ruizendaal, J., Hofstee, J. W., and van Henten, E. J. Fruit detectability analysis for different camera positions in sweet-pepper. *Sensors 14*, 4 (2014), 6032–6044. ⇒ 4

[11] Kapach, K., Barnea, E., Mairon, R., Edan, Y., and Ben-Shahar, O. Computer vision for fruit harvesting robots – state of the art and

challenges ahead. *Int. J. Comput. Vision Robot. 3*, 1/2 (Apr. 2012), 4–34. ⇒ 4

[12] Krizhevsky, A., Nair, V., and Hinton, G. The cifar dataset. [Online; accessed 27.10.2018]. ⇒ 2, 6

[13] LeCun, Y., Cortes, C., and Burges, C. J. The mnist database of handwritten digits. [Online; accessed 27.10.2018]. ⇒ 6

[14] Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning* (New York, NY, USA, 2009), ICML '09, ACM, pp. 609–616. ⇒ 8

[15] Li, D., Zhao, H., Zhao, X., Gao, Q., and Xu, L. Cucumber detection based on texture and color in greenhouse. *International Journal of Pattern Recognition and Artificial Intelligence 31* (01 2017). ⇒ 4

[16] Liang, M., and Hu, X. Recurrent convolutional neural network for object recognition. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015), pp. 3367–3375. ⇒ 6, 8

[17] Mumford, D., and Shah, J. Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics 42*, 5 (1989), 577–685. ⇒ 5

[18] Muresan, H., and Oltean, M. Fruits 360 dataset on github. [Online; accessed 27.10.2018]. ⇒ 1, 10, 27

[19] Muresan, H., and Oltean, M. Fruits 360 dataset on kaggle. [Online; accessed 27.10.2018]. ⇒ 1, 10

[20] Ninawe, P., and Pandey, M. S. A completion on fruit recognition system using k-nearest neighbors algorithm. In *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)* (2014), vol. 3. ⇒ 5

[21] O'Boyle, B., and Hall, C. What is google lens and how do you use it? [Online; accessed 05.05.2018]. ⇒ 2

[22] Puttemans, S., Vanbrabant, Y., Tits, L., and Goedem, T. Automated visual fruit detection for harvest estimation and robotic harvesting. In *2016 Sixth International Conference on Image Processing Theory, Tools and Applications (IPTA)* (Dec 2016), pp. 1–6. ⇒ 4

[23] Rahnemoonfar, M., and Sheppard, C. Deep count: Fruit counting based on deep simulated learning. *Sensors 17*, 4 (2017). ⇒ 4

[24] Ren, S., He, K., Girshick, R. B., and Sun, J. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*

abs/1506.01497 (2015). ⇒ 4

[25] Sa, I., Ge, Z., Dayoub, F., Upcroft, B., Perez, T., and McCool, C. Deep-fruits: A fruit detection system using deep neural networks. *Sensors 16*, 8 (2016). ⇒ 3, 4

[26] Schmidhuber, J. Deep learning in neural networks: An overview. *CoRR abs/1404.7828* (2014). ⇒ 5

[27] Selvaraj, A., Shebiah, N., Nidhyananthan, S., and Ganesan, L. Fruit recognition using color and texture features. *Journal of Emerging Trends in Computing and Information Sciences 1* (10 2010), 90–94. ⇒ 4

[28] Song, Y., Glasbey, C., Horgan, G., Polder, G., Dieleman, J., and van der Heijden, G. Automatic fruit recognition and counting from multiple images. *Biosystems Engineering 118* (2014), 203 – 215. ⇒ 3

[29] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. Striving for simplicity: The all convolutional net. *CoRR abs/1412.6806* (2014). ⇒ 6, 26

[30] Srivastava, R. K., Greff, K., and Schmidhuber, J. Training very deep networks. *CoRR abs/1507.06228* (2015). ⇒ 5

[31] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *CoRR abs/1409.4842* (2014). ⇒ 14

[32] TensorFlow. Tensorflow. [Online; accessed 05.05.2018]. ⇒ 3, 14

[33] Wikipedia. Convolution in mathematics. [Online; accessed 05.05.2018]. ⇒ 6

[34] Wikipedia. Deep learning article on wikipedia. [Online; accessed 05.05.2018]. ⇒ 5

[35] Wikipedia. Google lens on wikipedia. [Online; accessed 05.05.2018]. ⇒ 2

[36] Xiong, J., Liu, Z., Lin, R., Bu, R., He, Z., Yang, Z., and Liang, C. Green grape detection and picking-point calculation in a night-time natural environment using a charge-coupled device (ccd) vision sensor with artificial illumination. *Sensors 18*, 4 (2018). ⇒ 5

[37] Zawbaa, H., Abbass, M., Hazman, M., and Hassanien, A. E. Automatic fruit image recognition system based on shape and color features. *Communications in Computer and Information Science 488* (11 2014), 278–290. ⇒ 4