Tomato Leaf Disease Detection Using CNN

Köksal Kapucuoğlu
Electronic Engineer Department
Istanbul Technical University
Istanbul/TURKEY
kapucuoglu19@itu.edu.tr

***Abstract*— In this report, we describe the process of creating a model in which we can classify various diseases on tomato leaves using a convolutional neural network. We examine the effect of hyperparameters and layers used in forming a convolutional neural network on model education.**

I. Introduction

We are trying to automate high-accuracy processes in many areas, from high-level operations to our daily life, with studies on machine learning and computer vision. The agriculture and health sector is at the top of these fields.

Automation is very important in the agricultural sector. Sometimes there is a period of 1 year between the cultivation process of a product and the collection process. During this period, processes such as irrigation and spraying should be done correctly. Sick leaves may need to be pruned. Different spraying may be required for different types of disease. Sick products and healthy products must be separated at the stage of collecting products. In order for these processes to be done quickly, the system that will perform this process expects to know all this information. Here we will have given this information to the system using a convolutional neural network.

II. CNN Overview

Today, CNN is a model that we can apply to solve every image related problem. There were methods used for image problems before, but the main advantage of CNN compared to the previous ones is that it automatically detects important features without any human surveillance.

CNN is also advantageous in terms of calculation. It uses special convolution and pooling processes and performs parameter sharing. This makes CNN models work universally on any device, making it universally attractive. In this way, we will be able to use our model in automation systems.

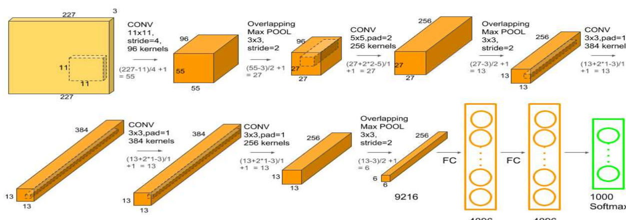

Fig. 1. Alexnet based architecture

In this work , we use a CNN architecture that was created based on Alexnet to create the model. Then, for a few questions that we seek answers, we create a model using Vgg16 with CNN architecture.

A. *AlexNet*

Alexnet has 8 layers. These are 5 convolutional layer and 3 fully connected layers. Input of Alexnet is 227x227x2 and output is 1000x1 so alexnet classify for 1000 classes. it has about 60 million parameters.

Multiple convolutional kernels extract intresting features in an image. In a single convolutional layer, there ara usually many kernels of the same size.

The first two convolutional layers are followed by the max poling layer. The third and fourth convolutional layers are connected directly. The fifth convolutional layer is followed by the max pooling layer. Then flatting is done. The second fully connected layer feeds into a softmax classifier for 1000 class.

ReLu nonlinearity is applied after all the convolutional and fully connected layers.

Max pooling layers are usually used to downsample the width and height of the tensors, keeping the depth same. At based Alexnet, max pooling layers use after first convolutional layer, second convolutinal layer and fifth convolutional layer. In Alexnet paper, the authors used pooling windows of size 3×3 with a stride of 2 between the adjacent windows.

B. *VGG16*

The VGG16 architecture consists of twelve convolutional layers, some of which are followed by maximum pooling layers and then four fully connected layers and finally a 1000 softmax classifier.

The 16 in VGG16 refers to it has 16 layers that have weights. This network is a pretty large network and it has about 138 million parameters.

III. Prepocessing

At this stage, we make our data ready to train with the CNN model we created. We decide the operations we will perform according to the dataset used.

A. *Detect Dataset and Get Data*

There is a problem we have identified. Linearly, the better we can introduce this problem to the system, the better our results will be. So the quality of the dataset we will use is very

important in this sense. This quality ensures both the size of the dataset and the variety of images for each class.

Here we use a dataset about "tomato leaf disease". This dataset has 22.930 tomato leaf images in total. 18345 images are used for train and 4585 images are used for validation. We also evaluate healthy tomato making as a class. In this way, there are a total of 10 classes and about 1800 images for each class. Each image is a 256x256 size RGB picture. All of these images are in JPG format.

First of all, in order to use these images, we have to import all the images as a array. It normalizes the values when taking images. The pixel values of the image are between 0 and 1. The reason we do this is related to numerical stability and convergence.

Apart from this, since we want to use the Alexnet based architecture, we should take our images that are 256x256 as 227x227. In addition, we will use 64/128/256 values as the batch size. Batch size is a hyperparameter that defines the number of samples to be worked on before updating internal model parameters. In this project 128 were used as batch size. In the training phase, instead of putting all the data into training one by one or all at once, we put it into training phase in smaller groups. In other words, the model trains 18345 images in each batch of training, about 144 times from 18345/128 operations in 128 batches. In addition to, it requires less memory.

B. *Data Augmentation*

I mentioned that the size of the data set and the variety of classes are required to get better results while training our model. We can use the "data augmentation" method when the opposite is a case, that is, when our data set gets little or if the images are very similar. Thus, we can expand our model data.

The only feature I definitely use here is the data rescale feature I mentioned in the previous section. In addition, in some cases, we have to use data augmentation. For example, in case of overfitting, we make our model more generalizable by using data augmentation.

The parameters and values that I use here; rescale = 1./255, width_shift_range = 0.2, height_shift_range = 0.2, shear_range = 0.2, zoom_range = 0.2, horizontal_flip = True, fill_mode = 'nearest'.

- Width Shifting: We can apply the width_shift_range technique to shift the image in the x direction.
- Height Shifting: We can apply the height_shift_range technique to shift the image in the y-direction.
- Shear Intensity: Here, we fix one axis and stretch the certain angle known as the shear angle. It stretches the image which is different than the rotation technique.
- Zoom Range: We can zoom the picture according to the zoom value here.
- Horizontal Flip: It flips the images horizontally by specifying the boolean value in the horizontal_flip parameter. By specifying true it flips them horizontally.
- Fill Mode – Nearest: In this mode, the closest pixel value is selected for all null values.

IV. TRAINING

At this stage, we need first create our CNN model. We use architecture that is based on Alexnet architecture. We need the image as the input and 10 classes as the output. Since we use supervising learning, that is, our data is labeled data, weights will be updated in every step of the training and we will try to find the best weights. In this way, we will create our generalized model.

A. *Baseline Model*

Alexnet classifies for 1000 classes, but our neural network needs to classify for 10 classes. Therefore, we first change the last layer "1000 softmax" layer to "10 softmax". Finally baseline model has about 58 million parameters. Apart from that, we do not change the base Alexnet model.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 55, 55, 96)	34944
max_pooling2d_1 (MaxPooling2	(None, 27, 27, 96)	0
conv2d_2 (Conv2D)	(None, 27, 27, 256)	614656
max_pooling2d_2 (MaxPooling2	(None, 13, 13, 256)	0
conv2d_3 (Conv2D)	(None, 13, 13, 384)	885120
conv2d_4 (Conv2D)	(None, 13, 13, 384)	1327488
conv2d_5 (Conv2D)	(None, 13, 13, 256)	884992
max_pooling2d_3 (MaxPooling2	(None, 6, 6, 256)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_1 (Dense)	(None, 4096)	37752832
dense_2 (Dense)	(None, 4096)	16781312
dense_3 (Dense)	(None, 10)	40970

Total params: 58,322,314
Trainable params: 58,322,314
Non-trainable params: 0

Fig. 2. Baseline Model for Tomato Leaf Disease Classifier

After creating the model, we proceed to the preprocessing process. At this stage, we set our entry image to 227x227 in accordance with the model entry. We also normalize the pictures when receiving. So we set the pixel values of the image are between 0 and 1.

To begin with, we set the batch size to 128. So baseline model will work on 128 training samples before the model's internal parameters are updated. This way, an epoch will consist of 144 steps. The average training time of an epoch is 45 seconds when a good GPU is used. With an average GPU, it is 240 seconds on average. Then we use my SGD and default SGD parameters as optimizers. Finally, to keep the total training time short, we initially set the number of epoch to 25.

In this way, as a result of the training process , we achieve train accuracy of 98.50% and validation accuracy of 92.25%.

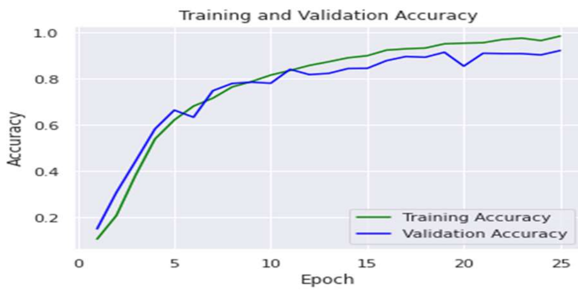

Fig. 3. Train and Validation Accuracy for Baseline Model

B. Optimization and Get Better Accuracy

According to our initial training result, we achieved a 98.50% train accuracy, while we achieved a lower validation accuracy of 92.25%. For a better train and validation accuracy, we can adjust hyperparameters and apply data augmentation to our data.

1) Initializer

In this project, we use the keras library for all these processes. In the Keras library, layers start with the "xavier initializer" by default. Xavier initializer is the weights initialization technique that tries to make the variance of the outputs of a layer to be equal to the variance of its inputs. Xavier initialization works better for layers with sigmoid activation. But we use ReLu activation function. In this case, it's better to use "He-uniform" initializer". He-uniform is known variance scaling initializer. He-uniform initialization works better for layers with ReLu activation.

When we use he-uniform initializer, we achieve train accuracy of 99.86% and validation accuracy of 95.17%.

2) Optimizer

In the beginning we used SGD as an optimizer. SGD is a variant of gradient descent. Instead of performing computations on the whole dataset, which is redundant and inefficient, SGD only computes on a small subset or random selection of data examples.

In the recent years, a number of new optimizers have been proposed to tackle complex training scenarios where gradient descent methods behave poorly. One of the most widely used and practical optimizers for training deep learning models is Adam.

When i use adam optimizer , i achieve train accuracy of 96.69% and validation accuracy of 92.07%. In this case, we see that i got a better result with SGD.

3) Regularziton

When we look at the trainings here, i reach 99.86% as train accuracy, but i reached the best 95% as validation accuracy. In this case, we need to look at other methods to get a better validation accuracy. The most important of these methods is the regularization process we will do to generalize the model.

a) Dropout

At each training iteration a dropout layer randomly removes some nodes in the network along with all of their incoming and outgoing connections. Dropout can be applied to hidden or input layer. The most commonly used dropout value is 0.5. That's why we chose 0.5 in the first training we used dropout.

When i use dropout(0.5), i achieve train accuracy of 92.32% and validation accuracy of 93.92%. In this case, i see that we got a better result without dropout(0.5).

	Dropout at fully connected layer	Dropout at conv layer	Train accuracy	Validation accuracy
Conf.1	0.5	-	%92.30	%93.92
Conf.2	0.5	0.2	%81.52	%77.07
Conf.3	0.4	-	%94.38	%91.43
Conf.4	0.2	-	%97.26	%94.95

Fig. 4. Train and validation accuracy of different dropout configurations.

b) L2 regularization

L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term. Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models.

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|^2$$

	L2 regularization	Train accuracy	Validation accuracy
Conf.1	0.1	%88.32	%90.37
Conf.2	0.01	%98.90	%94.40
Conf.3	0.05	%92.15	%90.66
Conf.4	0.001	%100	%95.21
Conf.5	0.0001	%99.9	%95.36

Fig. 5. Train and validation accuracy of different l2 configurations.

Here, lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero). I try L2 regularization in different parameters. I got the best result at 0.0001.

Fig. 6. Train and validation accuracy of different l2 configurations

c) Data Augmentation

Although we use L2 or dropout, there is still much difference between train accuracy and validation accuracy. Although training accuracy is 1, validation accuracy is at 95%. In this case, we can think that it is overfitting.

The simplest way to reduce overfitting is to increase the size of the training data or assuming that the data is uniform, we can increase the diversity for the class by taking some actions such as rotating, flipping and zooming in the pictures while taking the pictures. In Keras, we can perform all of these transformations using ImageDataGenerator.

When i use data augmantation, i achieve train accuracy of 95.55% and validation accuracy of 93.58%. In this case, i achieved a worse train and validation accuracy, but overfitting reduced. Here i prefer to use the model with data augmentation as it reduce overfitting.

4) Normaliztion

Normalization methods normalize each feature so that they maintains the contribution of every feature, as some feature has higher numerical value than others. This way our network can be unbiased.

a) Batch Normalization

Batch normalization is a normalization method that normalizes activations in a network across the mini-batch. For each feature, batch normalization computes the mean and variance of that feature in the mini-batch. It then subtracts the mean and divides the feature by its mini-batch standard deviation.

In the beginning, when I train using batch normalization and layer normalization, we did not use data augmentation. So we can see that the results are overfitting.

When i use batch normalization, i achieve train accuracy of 99.99% and validation accuracy of 95.74%.

b) Layer Normalization

Batch normalization normalizes the input features across the batch dimension. The base feature of layer normalization is that it normalizes the inputs across the features.

When we use batch normalization, i achieve train accuracy of 100% and validation accuracy of 95.05%.

5) Epoch

Up to this point, 25 epoches have been used in all of the trainings, but we may not have achieved the convergence we wanted in 25 epoch. Now let's train our model for 50 epoch and 100 epoch and look at the results.

Epoch	Data Augmentation	Train Accuracy	Validation Accuracy
50	-	%100	%95.21
50	+	%96.32	%95.51
100	+	%98.63	%97.78

Fig. 7. Train and validation accuracy of different epoch and data augmantation configurations

When i use 100 epoch in the configuration i use data augmentation, we achieve train accuracy of 98.63% and validation accuracy of 97.78%.

If we i 0.2 dropouts for the same configuration instead of l2 regularization, i achieve train accuracy of 97.76% and validation accuracy of 97.98%.

Fig. 8. Train and validation accuracy of final model.

V. Other Optimization Method

During training, we can use pre-trained models instead of training from scratch. Since pre-trained models are trained on their big data, it is better to learn discriminatory features than from scratch training.

I can only use the convolutional layers as a feature extractor or change the currently trained convolutional layers to suit our problem at hand. The first approach is known as “Transfer Learning”, the second is “Fine tuning”.

A. Transfer Learning

To implement the transfer learnig method, we need a model with a larger data trained using our alexnet architecture. For this, i use the model trained with a dataset related to plant leaves disease in Kaggle. The output of this model predicts for 38 classes. It contains 70295 images for Dataset traning and 17572 images for validation.

Here, after creating the model and loading the pre-trained weights, i remove last dense layer that classify 38 class and add the dense layer to classify it for 10 classes. When i do this and train the model again, i reach 97.75 as validation accuracy.

B. *Fine Tunning*

We can simply divide the layers in our model into two as feature extractors and classifiers. Training the model in a small dataset greatly affects the model's ability to generalize. For this reason, we can take trained weights in larger datasets and fine tune them. Here i only retrain fully connected layers, especially without touching extractive layers.

When i use batch normalization, i achieve train accuracy of 97.50% and validation accuracy of 97.84%.

C. *Vgg ile transfer learning*

Up to this stage, i have done all your operations on the Alexnet-based architecture. In order to get more efficiency from fine tunning process, it is necessary to take the pre-trained weights in datasets such as Imagenet containing 1.2M data.

We could not reach weights trained in such a large dataset for Aalexnet architecture. That's why we use the VGG16, where we can reach the weights of the model, which are both widely used and trained on Imagenet.

In the first stage, it is necessary to apply preprocess operations. I both apply data augmentation to the images and take the images in 224x224 size. I use 128 as batch size. In other words, 128 input samples will be used in each step and will be updated later by weight. After creating the layers of the VGG model, i load the trained weights at Imagenet.

When i train Vgg16 and pre-trained weight, i achieve train accuracy of 98.53% and validation accuracy of 98.55%.

VI. Conclusion

As a result, with the model i created, i reached a validation accuracy of 97.98%. We use 62,388,354 parameters in total.

Layer (type)	Output Shape	Param #
conv2d_26 (Conv2D)	(None, 55, 55, 96)	34944
activation_41 (Activation)	(None, 55, 55, 96)	0
max_pooling2d_16 (MaxPooling	(None, 27, 27, 96)	0
conv2d_27 (Conv2D)	(None, 27, 27, 256)	614656
activation_42 (Activation)	(None, 27, 27, 256)	0
max_pooling2d_17 (MaxPooling	(None, 13, 13, 256)	0
conv2d_28 (Conv2D)	(None, 13, 13, 384)	885120
activation_43 (Activation)	(None, 13, 13, 384)	0
conv2d_29 (Conv2D)	(None, 13, 13, 384)	1327488
activation_44 (Activation)	(None, 13, 13, 384)	0
conv2d_30 (Conv2D)	(None, 13, 13, 256)	884992
activation_45 (Activation)	(None, 13, 13, 256)	0
max_pooling2d_18 (MaxPooling	(None, 6, 6, 256)	0
flatten_6 (Flatten)	(None, 9216)	0
dense_21 (Dense)	(None, 4096)	37752832
activation_46 (Activation)	(None, 4096)	0
dropout_1 (Dropout)	(None, 4096)	0
dense_22 (Dense)	(None, 4096)	16781312
activation_47 (Activation)	(None, 4096)	0
dropout_2 (Dropout)	(None, 4096)	0
dense_23 (Dense)	(None, 1000)	4097000
activation_48 (Activation)	(None, 1000)	0
dropout_3 (Dropout)	(None, 1000)	0
dense_24 (Dense)	(None, 10)	10010

Total params: 62,388,354
Trainable params: 62,388,354
Non-trainable params: 0

Fig. 9. Layer of final model.

Fig. 10. Train and validation accuracy for final model.

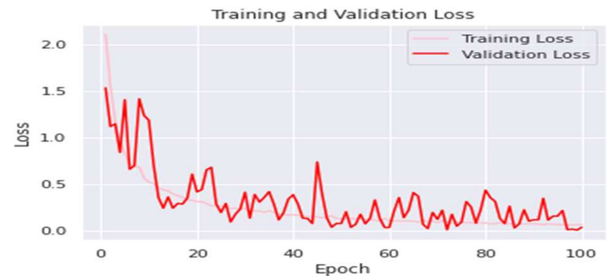

Fig. 11. Train and validation loss for final model.

VII. What I have learned and contributed

First of all, the most important factor in a model education is data. It is very important that we have a large number of data and a large variety of data. For example, while training a model from scratch, we get a validation accuracy of around 95%, while we can achieve 98% validation accuracy in the model trained with about 88 thousand data. In order to achieve the same validation accuracy, we should use 100 epoch in the model we train from scratch. This increases the training cost.

Hyperparameters need to determine correctly. A lot of testing should be done for this. According to the test results, the behavior of the data should be examined and the correct hyperparameter values should be given.

References

[1] https://www.tensorflow.org/api_docs/python/tf/keras/initializers/he_uniform

[2] https://www.dlology.com/blog/quick-notes-on-how-to-choose-optimizer-in-keras/

[3] https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/

[4] https://mlexplained.com/2018/01/13/weight-normalization-and-layer-normalization-explained-normalization-in-deep-learning-part-2/

[5] https://towardsdatascience.com/backpropagation-and-batch-normalization-in-feedforward-neural-networks-explained-901fd6e5393e

[6] https://chatbotslife.com/regularization-in-deep-learning-f649a45d6e0

[7] https://www.machinecurve.com/index.php/2019/12/18/how-to-use-dropout-with-keras/

[8] https://flyyufelix.github.io/2016/10/03/fine-tuning-in-keras-part1.html

[9] https://www.learnopencv.com/keras-tutorial-transfer-learning-using-pre-trained-models/