[Team 21] Project 1.2: Leaf Wilting Detection in Soybean

Daniel Mendez Lozada
dmendez@ncsu.edu

Alhiet Orbegoso
aoorbego@ncsu.edu

Chanae Ottley
cottley@ncsu.edu

I. Methodology

For this project, we chose to classify images of healthy and wilted soybean leaves using a Convolutional Neural Network (CNN). We performed transfer learning by adapting the VGG16 [1] model to the soybean data.

Code development was made using *Google Colab* and can be reviewed in this GitHub repository [3]. It consists of three scripts: *Dataset.ipynb, VGG16.ipynb* and *Results.ipynb*. The first organizes the data in directories, the second is where the model is trained, and the last one makes the predictions.

For the pre-processing stage, we performed dimensionality reduction and data augmentation. Each image was reduced from 480x640 pixels to 224x224 pixels using methods from the *cv2* library. We used the *ImageDataGenerator* class from Keras to balance the data sets.

For our final architecture, we chose the VGG16 CNN as the base, which achieves 92.7% top-5 accuracy in the ImageNet dataset. It consists of 13 convolutional layers and 3 dense layers on top, and it resulted in a good fit given the classification nature of the project. See Figure 1 for details regarding the number of neurons and pooling layers.

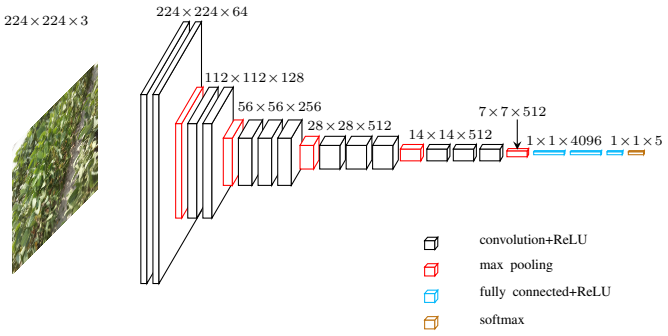

Fig. 1. CNN architecture.

II. Model Training and Selection

A. Model Training

This project's provided data [5] is small and unbalanced, with a total of 1275 images divided into 488, 329, 130, 131, and 197 images per class, respectively. The class imbalance represents a challenge since we have to account for this during training.

First, we created a *pre-test set* of 100 samples comprised of 20 images of each class. From the remaining 1175 images, we followed the *30-70 rule* and split our data into training and validation sets with 850 and 350 images, respectively. Here, we created a balanced validation set with 70 images in each class. Finally, after splitting our data into these three sets, we end up with a training data set with a count of 398, 239, 40, 41, and 107 in each class, respectively. Figure 2 shows the data distribution.

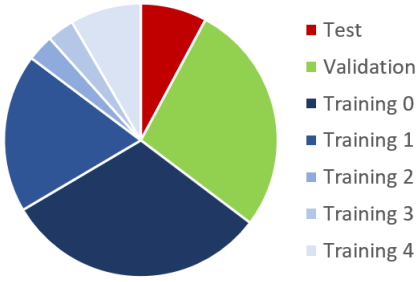

Fig. 2. Data Distribution.

We trained our model to account for dissimilarities in class distribution by applying different weights to our loss function. With different weights for each class, we can penalize those sets more with less data during training.

To compute the weights, we used the utility *computeclassweight* from *sklearn*, which returns a dictionary mapping the classes and its weights. Normalizing the weights before fitting the model is not necessary according to the *keras.fit* documentation.

$$0 : 0.41, 1 : 0.69, 2 : 4.125, 3 : 4.02, 4 : 1.54$$

To further balance the data set, we used *data augmentation*. However, this concept may be misinterpreted: one may assume that data augmentation is equivalent to increase the training samples and store both the original and augmented samples. However the concept of data augmentation is not to increase the samples available for training, more strictly, the goal of data augmentation is to increase the generalizability of the model by replacing the training samples with new data-augmented images.

We used the *ImageDataGenerator* class from Keras [6] to perform data augmentation using different data augmentation schemes: horizontal flip, image zoom, rotation and brightness

adjustment. Figure 3 shows 9 augmented images and their respective one-hot labels. Note that data-augmentation schemes are applied at random.

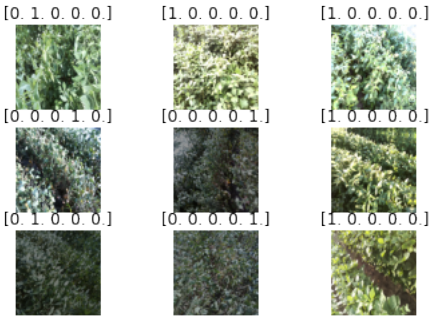

Fig. 3. Data Augmentation.

Using an instance of the *ImageDataGenerator* we are able to generate a whole new set of images each epoch during training. Using this class has the benefit of training the model *on-the-fly*, i.e. temporarily generating a new batch of images for training. This reduces memory requirements and allows the model to generalize since it is trained with a non-static set of images.

B. Model Selection

Given the base-line results of *project 1.1*, we noticed that the top-5 best results were achieved by means of *transfer learning*. This lead us to use one of most popular networks for transfer learning: VGG16.

Originally, we design our own convolutional network based on the AlexNet [4], however we reached only an F1 score of 21 in the first round and thus decided to try transfer learning using two main models: ResNet-18 and VGG16. Within the first training attempts and under the same testing scenarios we noticed that VGG16 perform better. In Figure 4 we show one failed attempt of training using the ResNet model.

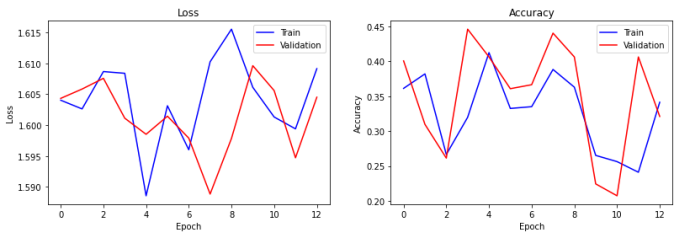

Fig. 4. Loss and accuracy of training using the ResNet model.

The final model consisted on taking the convolutional layers of the VGG16 model and adding 3 fully-connected layers on top. Following the transfer learning methodology, we froze the convolutional layers and trained only the dense layers on top to preserve the trained weights. Furthermore, we performed *fine tunning* by unfreezing the whole model and training it for a few epochs and with a low learning rate.

During training, we used *early stopping* to avoid over-fitting and *model checkpoint* to save the best model parameters. In this architecture, the only hyper-parameter is the learning rate. To optimize the model, we ran a *grid search* with cross-entropy as the loss function, performed 5 folds, and searched for the best parameters using learning rates from 0.1 to 0.0005. After running the search, we found the best parameter to be a value a learning rate of 0.001.

III. Evaluation

The final model had as best accuracy of 0.7% and as best loss of 0.72, both in the validation set. Figures 3 and 6 show the respective plots.

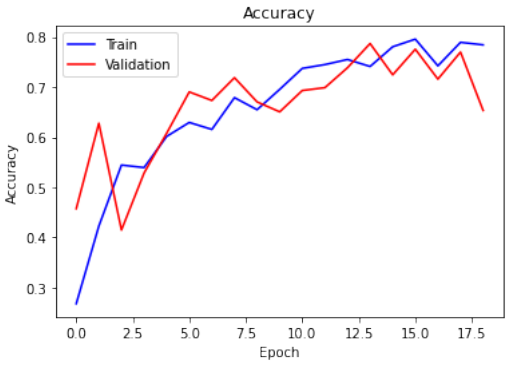

Fig. 5. Final model accuracy

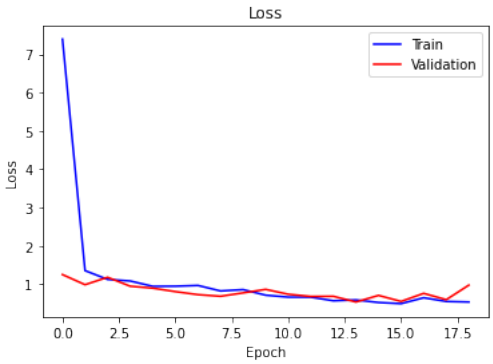

Fig. 6. Final model loss

The base model used as a start did not considered any fine tuning step. After fine tuning is observed an improvement in accuracy with a value of 0.79% and in loss with a value of 0.55. Figures 7 and 8 present accuracy and loss of the neural network with fine tuning.

Finally, we measure performance of the CNN using the *precision*, *recall*, *accuracy* and *F1*. Equations 1 to 4 are used to compute these results and are summarized in Table I and in Figure 9 for validation data and in Table II and Figure 10 for pre-test data. For both datasets the fine tuned model is used.

Even though our results suggest a good performance, the highest F1 score we achieved for submission was 0.49. This

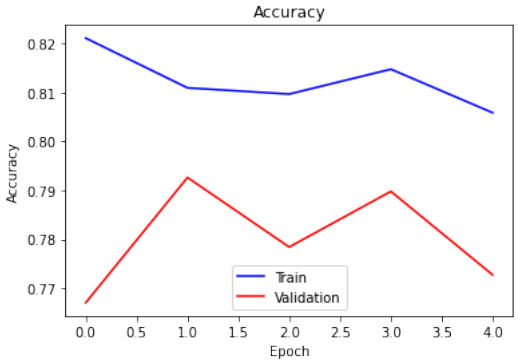

Fig. 7. Fine tuning accuracy

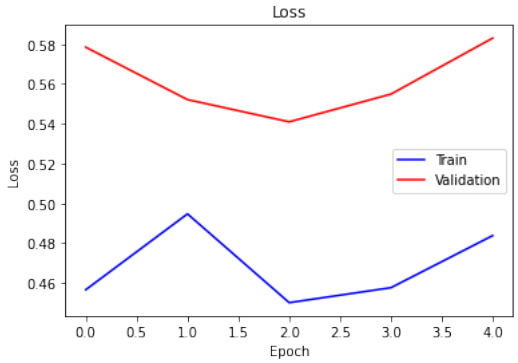

Fig. 8. Fine tuning loss

suggests that the data on which the score was based, might be unbalanced causing the score to drop significantly.

$$precision = \frac{tp}{tp + fp} \tag{1}$$

$$recall = \frac{tp}{tp + fn} \tag{2}$$

$$accuracy = \frac{tp + tn}{tp + tn + fp + fn} \tag{3}$$

$$F_1 = 2\frac{precision \cdot recall}{precision + recall} \tag{4}$$

TABLE I
CLASSIFICATION RESULTS VALIDATION DATASET

class	precision	recall	f1-score
0	0.87	0.84	0.86
1	0.87	0.76	0.81
2	0.78	0.73	0.76
3	0.75	0.94	0.84
4	0.99	0.96	0.97

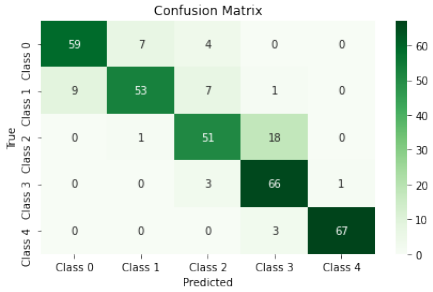

Fig. 9. Validation Confusion Matrix

TABLE II
CLASSIFICATION RESULTS PRE-TEST DATASET

class	precision	recall	f1-score
0	0.83	0.75	0.79
1	0.89	0.80	0.84
2	0.63	0.60	0.62
3	0.67	0.90	0.77
4	1.00	0.90	0.95

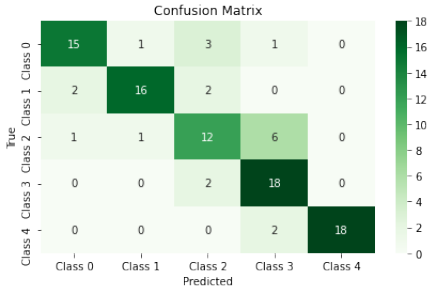

Fig. 10. Pre-Test Confusion Matrix

REFERENCES

[1] Simonyan, K. Zisserman, A. *Very Deep Convolutional Networks for Large-Scale Image Recognition* arXiv preprint arXiv:1409.1556.

[2] K. Simonyan A. Zisserma, *"Very Deep Convolutional Networks for Large-Scale Image*. Recognition".Oxford University.

[3] Project repository: https://github.com/danmenloz/LeafWilting

[4] Krizhevsky A., Sutskever I. and Hinton G., *ImageNet Classification with Deep Convolutional Neural Networks*

[5] Origianl dataset: https://drive.google.com/drive/folders/1-3r7Hvt2MpLzO-ZKkSqwXKUIZtAMQKLL

[6] Data Augmentation with Keras: https://machinelearningmastery.com/image-augmentation-deep-learning-keras/