Capstone Project

Allyson Julian

February 8, 2020

1 Definition

1.1 Project Overview

Advances in agricultural technology over the past 30 years have made it easier for farmers to manage their farms, particularly when the farms are comprised of multiple fields greater than 1000 acres in size. The use of GPS on aerial imagery, drones, etc. have been instrumental in precision agriculture [1].

But one of the challenges of modern farming is the detection of plant diseases. For large farms in particular, it is time-consuming for a farmer to manually check each growing plant for disease. It can potentially be more cost-effective to diagnose plant diseases with automated tools [2].

For this project, I built the Python library **ikapati** (named after the Filipino goddess of agriculture), which implements a plant disease detector and then deployed the best-performing model to an embedded system built specifically for use in Artificial Intelligence applications, a Jetson Nano robot.

The library also provides utility tools to do many of the tasks required in machine learning research, including the image preprocessing necessary to prepare the data for classification, scripts to train classifier itself, and utility functions to help evaluate performance.

1.2 Problem Statement

The main objective for this project was to build a library that can be used for training a plant disease classifier and in doing so act as a framework with which to do further research in this subject area. It can be used to complete an entire machine learning pipeline from training to deployment to an embedded system.

To build this library, I needed to:

1. Retrieve the PlantVillage Dataset (`https://github.com/spMohanty/PlantVillage-Dataset`).
2. Prepare the raw color images from the PlantVillage Dataset for consumption by the model.
3. Train the model to classify the different diseases for a given species of plant.
4. Save the model as a TensorFlow Lite object for use in an embedded system.

1.3 Metrics

Accuracy and loss were the main metrics used in this project to evaluate overall model performance.

Accuracy is generally defined as such:

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Where TP = true positives, TN = true negatives, FP = false positives, FN = false negatives.

Validation and training loss was used to determine whether the model was overfitting, underfitting, or fitting adequately.

If the curve of the validation loss decreases until a certain point, and then starts to increase again, it is indicative of overfitting.

If the curve of the training loss remains flat then the model may be underfit.

2 Analysis

2.1 Data Exploration

The main datasets used were obtained from the PlantVillage Dataset: `https://github.com/spMohanty/PlantVillage-Dataset`

The dataset consists of 52,803 JPEG images with the dimensions 256x256. All of the images are colored, so there are 3 channels corresponding to the RGB mode.

The images are cropped in such a way that the leaves are centered in the picture, and have a fairly uniform-looking, solid background.

The images are split up into several folders, labelled according to the plant species (e.g. Peach) and disease class (e.g. Bacterial spot).

(a) Corn leaf spot (b) Pepper (c) Apple Scab

Figure 1: Example images in dataset.

Further data exploration can be viewed in the 1.0-agj-data-exploration.ipynb notebook.

2.2 Algorithms and Techniques

A **Convolutional Neural Network (CNN)** was selected as the classification algorithm due to its demonstrated efficacy in plant disease detection in previous studies [3] [2].

Two CNN architectures were used in this project:

1. is based on **AlexNet** [4] and is comprised of Conv2D (which does convolution), MaxPooling2D (max pooling), and FCD (which are a combination of Dense and Dropout layers).

2. is based on the **InceptionV3** based network built in the study by

Toda et al [3] that demonstrated an accuracy of 99.99% in predicting plant disease classification on the PlantVillage Dataset.

2.2.1 AlexNet

This describes the layer sequence (the number indicates how many of that same layer repeats until the next one):

- Conv2D (3) - Convolution Layer
- MaxPooling2D (1) - Max Pooling Layer
- Conv2D (2) - Convolution Layer
- MaxPooling2D (1) - Max Pooling Layer
- FCD (3) - Fully Connected Layer w/ optional Dropout.

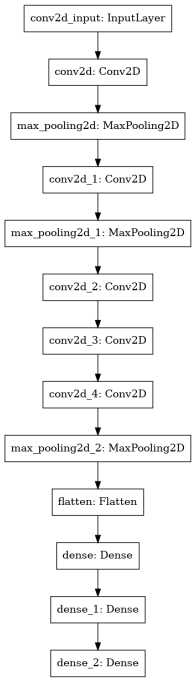


Figure 2: AlexNet Architecture

2.2.2 InceptionV3

This InceptionV3-based network is based on what was done in a previous study [3], but as 10 Mixed modules instead of the 11 used in that study.

- Conv2D (3) - Convolution Layer with Batch Normalization
- MaxPooling2D (1) - Max Pooling Layer
- Conv2D (2) - Convolution Layer with Batch Normalization
- MaxPooling2D (1) - Max Pooling Layer
- Mixed (10) - Sets of Conv2D with batch normalization
- GlobalAveragePooling2D (1) - Global Pooling Layer
- Dense (1) - The Output Layer

See **Additional Figures** to view visualization of InceptionV3.

2.2.3 Hyperparameters

These parameters can be tuned/specified at training time:

- **epochs** - the number of times to run through the training set.
- **learning rate** - the learning rate for the Adam optimizer.
- **batch size** - the number of training examples in each batch.
- **activation** - the activation function to use, e.g. "relu".
- **dropout** - the dropout rate, e.g. 0.2. This was only supported by AlexNet.
- **architecture** - the architecture to use for the CNN, "alexnet" (for AlexNet) or "inceptionv3" (for InceptionV3).

2.3 Benchmark

The benchmark for the plant disease classifier are results obtained by previous studies by Toda et al, Fuentes et al, on plant detection which all utilize a CNN as a classifier[3] [5].

3 Methodology

3.1 Data Preprocessing

Preprocessing the data was completed using these steps:

1. Get a list of all image filenames.
2. Shuffle list of image filenames.
3. Get labels from the image folder names. For example: images that were in the Apple___Apple_scab were assigned the label Apple___Apple_scab.
4. Split list of image filenames into training files (60%), validation (20%), and testing files (20%) to mirror the split in a previous study [3] .
5. Create a training example for each image file so it can be added to a TFRecord (TensorFlow dataset).
6. Write metadata describing the file counts of training, validation, and test sets and list the class names.
7. Create parser function to read each TFRecord batch by batch during training due to hardware limitations (limited VRAM, in this case).
8. Normalize image pixel values by dividing by 255 (representing the range of values for an RGB image) and subtract -0.5 to the values to make them fall in the -0.5 to 0.5 range.

3.2 Implementation

The library was primarily written in Python 3. TensorFlow 2.0 with the keras API was used in the training stage.

The modules for the **ikapati** library can be broken down as such:

- **models** - these contain the training script and code to build the networks.
- **data** - this has the utility functions used to preprocess the data, e.g. normalizing pixel values, reading datasets into memory.
- **visualization** - this contains utility functions to help visualize and evaluate model performance. Figures included in this report were generated using that module.

3.2.1 Data Preparation

The datasets were prepared using the **ikapati/data/make_dataset.py** script (which utilizes datasets scikit-learn and TensorFlow 2.0) and then saved as TFRecords, a format used by TensorFlow.

1. Get the filenames of all the images that match the specified plant species.
2. Follow the steps outlined in the **Data Preprocessing** section.
3. Put each subset of training examples (training, validation, and test) into separate TFRecords (train.tfrecord, eval.tfrecord, test.tfrecord).

3.2.2 Training Stage

The training models were created using TensorFlow 2.0 with the keras functional API. As described in the section **Algorithms and Techniques**, CNN was chosen as the classifier, with neural network architectures based on AlexNet and InceptionV3 as described in a previous study [3] . TensorFlow was configured to use a physical GPU similar to what was used in the Toda study, a GeForce GTX 1080 Ti with 11GB of VRAM, to do the training.

The code used to build the AlexNet architecture was based on:

- `https://engmrk.com/alexnet-implementation-using-keras/`
- `https://github.com/tensorpack/benchmarks/blob/master/other-wrappers/keras.alexnet.py`

The code used to build the InceptionV3 architecture was based on:

- `https://github.com/keras-team/keras-applications/blob/master/keras_applications/inception_v3.py`

The training steps for each training run were:

1. Execute the ikapati/models/train_model.py script with the architecture (alexnet or inceptionv3), learning rate, epochs, batch size, activation, and dropout rate (if architecture is set to alexnet) hyperparameters specified, the model and data directories set, and checkpoint saving enabled to kick off a training run. (See the **Hyperparameters** described in the section **Algorithms and Techniques** for more details on the hyperparameters).

2. Create a folder under the model dir specified at runtime, with the UUID of dataset used as the name, and a subfolder within that with the start time string as the name. This will be where the models are saved during this training run.

3. Record the start time.

4. After each epoch, if the validation loss has improved from the previous epoch, save the model to file in h5 format. Otherwise, we don't save anything and proceed with the next epoch.

5. When all epochs have concluded, save the final model to file and record the end time.

6. Write the start and end time of the training run, the learning rate, and other parameters specified at runtime to a CSV file that acts as a log of training runs.

7. Repeat steps above, tweaking the hyperparameters as needed.

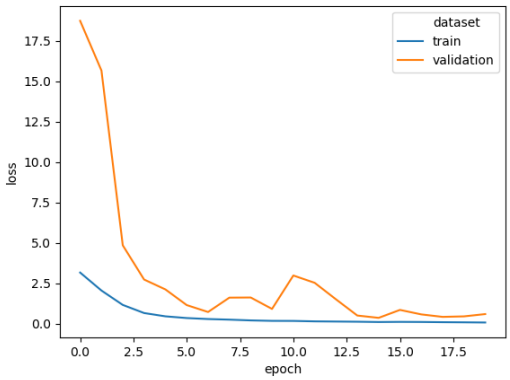
3.3 Refinement

In the initial benchmark run, the hyperparameters were set to match that of the benchmark model from Toda et al, with the learning rate set to 0.05, the batch size to 128, and the epochs to 20. The architecture was inceptionv3 and the activation function was "relu" (rectified linear units). **NOTE** The benchmark model input shape was 224x224, where as in this project the input shape was 256x256.

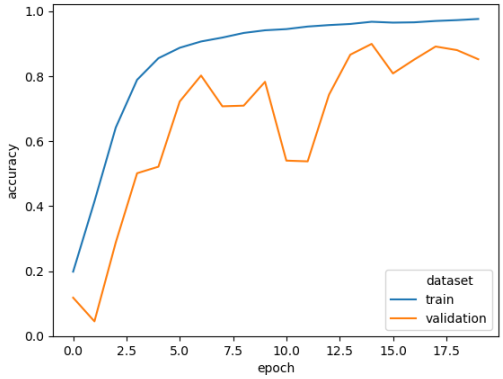
The benchmark model achieved an accuracy of 85.68% on the test dataset, and a loss score of 0.570 (Figure 8).

In subsequent training runs, certain modifications were made to the hyperparameters to see if it would improve the model validation loss and accuracy:

- The learning rate was decreased to 0.001.
- The batch size was reduced to account for hardware limitations.
- The epoch number was increased to see if the model's validation loss decreases over time.
- If validation loss improved in one epoch, then a checkpoint of the model was saved to file.



(a) Training vs Validation Loss



(b) Training vs Validation Accuracy

Figure 3: Loss and accuracy plots for the benchmark model.

- After each iteration, the validation vs training loss was plotted to determine whether a model is underfitting or overfitting, or reaching a good fit.
- When selected as the architecture, AlexNet demonstrated a tendency to overfit on this dataset even with dropout layers, so InceptionV3 became the defacto architecture.

3.3.1 Visualization and Metrics Tools

Utility functions to evaluate model performance were created using seaborn (a high-level API for matplotlib) and pandas.

3.3.2 Deployment to an Embedded System

Upon training completion, trained models as saved in both h5 and tflite (TensorFlow Lite) formats. Both formats can be loaded for later use with the TensorFlow library, but the tflite format is specifically used to run inference on embedded devices such as the Jetson Nano mentioned previously.

A simple web app was created using the Flask framework.

This web app has a POST endpoint located at `http://jetbot:5000/predict`, which loads the tensorflow lite model and then attempts to predict the class name for an input image uploaded using that endpoint.

To simulate a POST request to that endpoint, the curl commandline tool was used with the image file passed in:

```
curl -F "image=@blight.jpg" http://jetbot:5000/predict
```

This endpoint expects a 256x256 image. If the request is successful, the result will be a JSON body containing the predicted class for the input image:

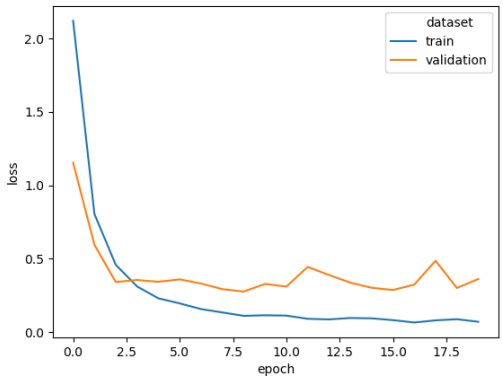
```
{
  "class_name": "Tomato___Septoria_leaf_spot"
}
```

4 Results

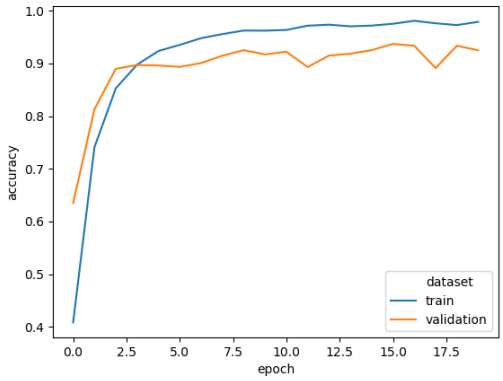
4.1 Model Evaluation and Visualization

Model performance was evaluated with accuracy and loss plots (Figure 5).

Models that showed overfitting, such as the AlexNet-based model, were discarded.



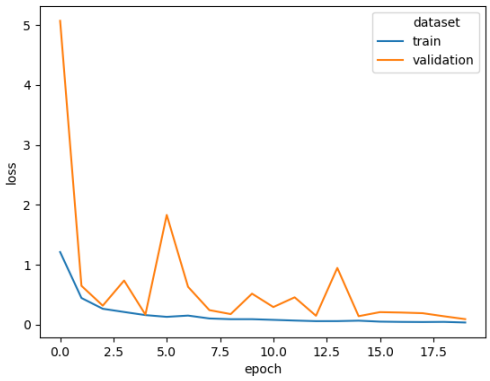
(a) Training vs Validation Loss



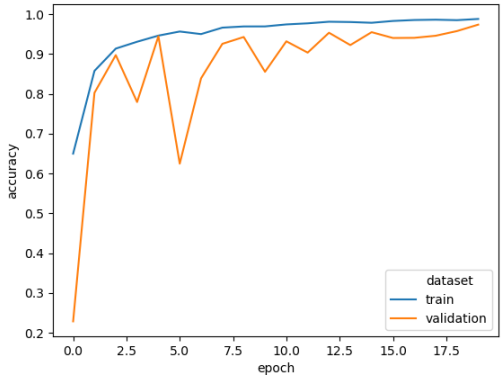
(b) Training vs Validation Accuracy

Figure 4: Loss and accuracy plots for AlexNet. Validation shows overfitting.

The 2 best performing models, based on these metrics, were both InceptionV3-based and outperformed the benchmark. In both of these runs, the learning rate was reduced to 0.001 from the benchmark 0.05.



(a) Training vs Validation Loss

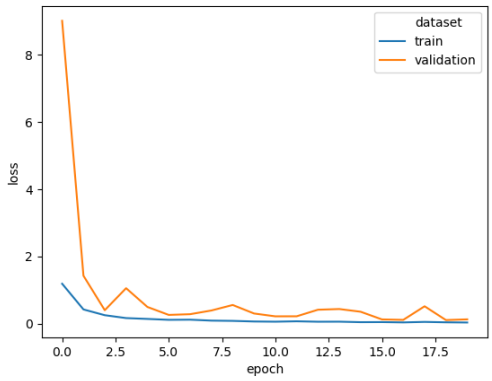


(b) Training vs Validation Accuracy

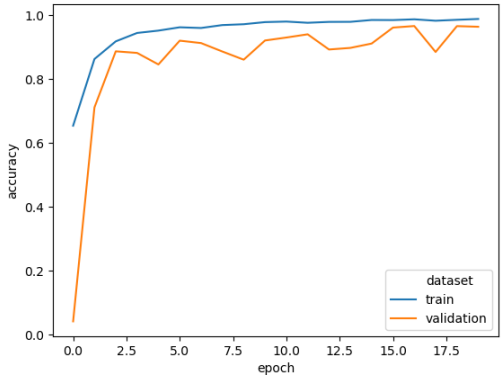
Figure 5: Model 1. Loss and accuracy plots for InceptionV3, with batch size of 64 and learning rate of 0.001.

In Figure 5, the plots show how that training run Model 1, with the batch size reduced to 64 from 128 performed on the validation and training sets. On the test dataset, this model performed with an accuracy of 97% and a loss score of 0.089.

In Figure 6, the plots show how the training run with the same batch size as the benchmark but a learning rate reduced to 0.001 performed. On the test dataset, this this model performed with an accuracy of 96% and a loss score of 0.126.



(a) Training vs Validation Loss



(b) Training vs Validation Accuracy

Figure 6: Model 2. Loss and accuracy plots for InceptionV3, with batch size of 100 and learning rate of 0.001.

4.2 Justification

While Model 1 and 2 performed similarly on the training and validation datasets, Model 1 did better on the test dataset than Model 2, so it was selected as the best model.

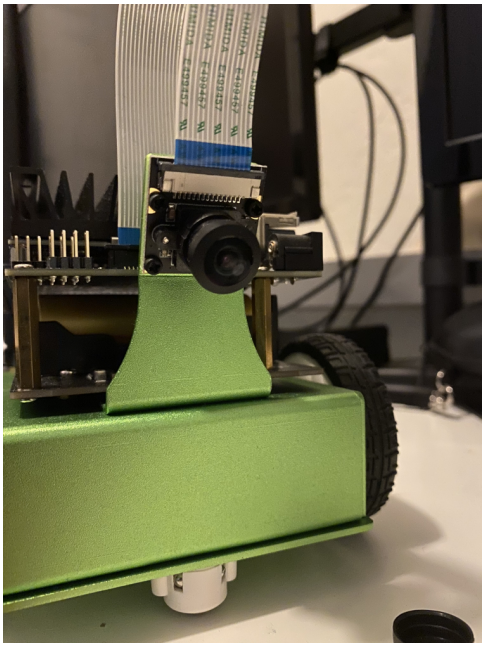
With an accuracy of 97%, and a loss score of 0.089, Model 1 also outperforms the benchmark model, which had 85.68% accuracy and a loss score of 0.570 on the test data.

Model 1 was converted to TensorFlow Lite format (tflite) and then trans-

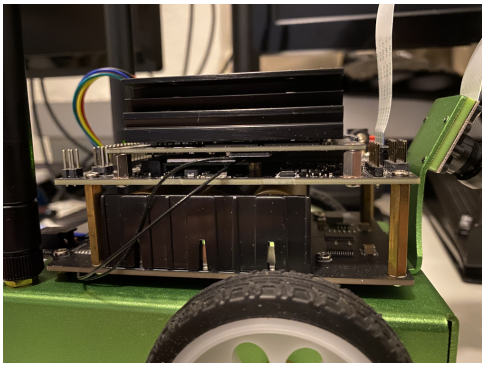
ferred to the Jetson Nano Robot. The Flask app loaded the tflite model, and then listened for any requests to the `/predict` endpoint.

5 Conclusion

5.1 Free-Form Visualization



(a) Jetson Nano Robot with Camera



(b) Side view of the Jetson Nano Robot

Figure 7: Jetson Nano Robot

(a) Strawberry leaf spot (b) Septoria leaf spot (c) Common corn rust

Figure 8: Images the classifier failed on. [6] [7] [8]

5.2 Reflection

These steps summarize what was done to complete this project:

1. Initial exploration of the agricultural domain was done to see what problems need more investigation.
2. A large dataset with multiple classes of plants was downloaded and then preprocessed.
3. A benchmark model was created based on a previous study [3] .
4. Models built in TensorFlow were trained using different hyperparameters and 2 different architectures.
5. The best model was chosen using validation loss and accuracy scores and then converted to TensorFlow Lite format.
6. A simple Flask API that loads the TensorFlow Lite model and does inference, was deployed to a Jetson Nano.

I found 3-4, and 6 most challenging because I had never used TensorFlow at the level this project required and had never built a CNN before.

It was interesting to learn about how those neural networks work and how they can be improved upon.

5.3 Improvement

There is some opportunities for improvement from this project:

- The dataset used here consisted of pictures with leaves that were set against a fairly uniform background. In a future iteration of this project, the dataset could be augmented by introducing random noise and transformations to a subset of images, and then use that to train.
- The machine learning model could be converted to do online learning so that learning can be done as inference is attempted. This could be especially useful when trying to improve the accuracy of the model.

References

[1] K. G. Liakos, P. Busato, D. Moshou, S. Pearson, and D. Bochtis, "Machine Learning in Agriculture: A Review," *Sensors*, vol. 18, no. 8, p. 2674, Aug. 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/8/2674

[2] E. Fujita, Y. Kawasaki, H. Uga, S. Kagiwada, and H. Iyatomi, "Basic Investigation on a Robust and Practical Plant Diagnostic System," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. Anaheim, CA, USA: IEEE, Dec. 2016, pp. 989–992. [Online]. Available: http://ieeexplore.ieee.org/document/7838282/

[3] Y. Toda and F. Okura, "How Convolutional Neural Networks Diagnose Plant Disease," 2019. [Online]. Available: https://spj.sciencemag.org/plantphenomics/2019/9237136/

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3098997.3065386

[5] A. Fuentes, S. Yoon, S. C. Kim, and D. S. Park, "A Robust Deep-Learning-Based Detector for Real-Time Tomato Plant Diseases and Pests Recognition," *Sensors*, vol. 17, no. 9, p. 2022, Sep. 2017. [Online]. Available: https://www.mdpi.com/1424-8220/17/9/2022

[6] "strawberry leaf spot (Mycosphaerella fragariae)," Apr. 2008. [Online]. Available: http://www.invasive.org/browse/detail.cfm?imgnum=5365398

[7] "Septoria leaf spot (Septoria helianthi)," Feb. 2011. [Online]. Available: http://www.invasive.org/browse/detail.cfm?imgnum=5430722

[8] "common corn rust (Puccinia sorghi)," Mar. 2012. [Online]. Available: http://www.invasive.org/browse/detail.cfm?imgnum=5465560