CZ4041 Machine Learning Project Report

Bryan Leow Xuan Zhen   Feng Daifei   Kong Jie Wei   Wang Xiaoyu   Wu Wenxuan

Contents

1 Introduction

Image classifications is one of most important topics in modern machine learning/deep learning research. It also poses many interesting yet difficult challenges to the academic community. With the increasing number of images available nowadays, processing images and finding patterns in them become possible with the help of the increasingly powerful processing units (e.g. GPU). Among the list of the Kaggle competitions, our group thus has identified an intellectually stimulating image classification problem - classifying leaves.

1.1 Problem Statement

The goal of the project is to identify specie of plants using either a plant leaf's binary image or using features extracted from the plant leaf image (i.e. shape, margin, texture). We aim at successfully establish multiple models that can accurately identify the specie of a leaf. The ranking in Kaggle leaderboard is used as our metrics for success.

1.2 Challenges of the problem

1. Interpretation of the feature vectors

In most other classification problems, the features have concrete physical meanings and can be interpreted easily. In such cases, some human judgment can be made in selecting and pre-processing the features.

However, in this Kaggle competition, the feature vectors are extracted based on domain-specific feature engineering methods such as Gabor filters. As a result, without such domain knowledge, we are unable to interpret and make appropriate data processing.

2. Competitive leaderboard

As a well-publicized Kaggle competition, this classification problem attracted 1598 participants with over 5000 submissions. The top submissions typically have very fine-tuned models such that the log loss, which is the evaluation criterion, is 0.00000. While it seemed easy to achieve good prediction results during our first few experiments, it has become increasingly difficult when we tried to reduce the log loss to below 0.03.

3. Difficulty in selecting the best model

Due to the randomness in the train-test split, we may build our models based on different trainsets. However, this may produce different models using the same algorithms, hence different accuracies and log losses.

Furthermore, models such as Multi-Layer Perceptron have random initial weight settings, which may lead to different final weights to achieve local minimum loss.

In this classification problem, we realized that the final accuracies and log losses of the various models developed are quite similar (~ 0.01). Due to the randomness mentioned above, it is hard to decide which model performs better, i.e. has lower log loss, because the slight difference could be just due to a different seed being used.

1.2.1 Specific Roles

Bryan Leow Xuan Zhen	Feng Daifei	Kong Jie Wei	Wang Xiaoyu	Wu Wenxuan
Developer & Technical Writer	Developer & Technical Writer	Team Lead	Developer & Technical Writer	Researcher

2 Data Description

For this competition, there are 2 sets of data available. The first set of data consists of 1584 images of 99 species of leaves, with each species consisting of 16 samples. The images are the binary black leaves against white background. For each of the leaf image, there are 3 sets of features for each image:

- **Shape:** A contiguous descriptor
- **Texture:** Describes the interior texture histogram
- **Margin:** Describes the fine-scale margin histogram

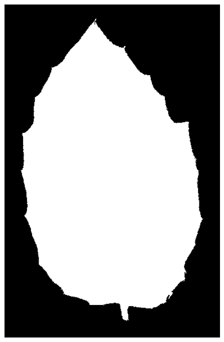
For each of these 3 features, there are 64 attributes, giving us a total of 192 features for each leaf.

This data set actually originates from Mallah's 2013 paper "*Plant Leaf Classification Using Probabilistic Integration of Shape, Texture and Margin Features*" (Mallah, Cope, and Orwell 2013). The details of how these features are obtained are not discussed in this report. Nevertheless, this paper has helped us tremendously in understanding the nature of data.

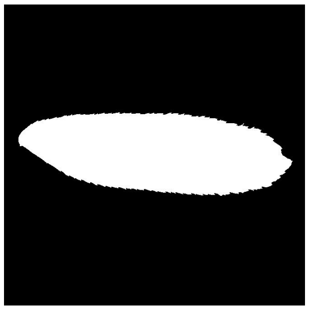
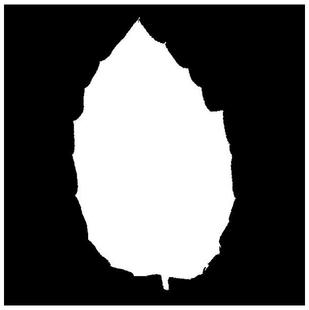
3 Pre-Processing

3.1 Image Data

Observing the image dataset given, it can be quickly observed that the images are of various sizes. Below are some of the images given.

Image 2	Image 4
	
(526 × 467)	(667 × 251)

Since our problem is to classify the dataset into the 99 species of leaves, a Convolutional Neural Network is a potential solution to classify the images. Therefore, image pre-processing is required for this dataset, to transform the images to a square, which allows us to easily resize and feed them into the Convolutional Neural Network. We performed image pre-processing by adding **black** padding to the shorter side of the images, such that they match the dimension of the longer side of the image, as shown below.

Pre-processed Image 2	Pre-processed Image 4
	
(526 × 526)	(667 × 667)

3.2 Other Features

For the other features, they are in the data type `float`, but of various scale. Feature Scaling have to be performed before running further machine learning algorithms so that there will not be a case where one feature is dominant compared to other features just because of the difference in magnitude. We performed feature scaling on these features by normalizing the features, using `StandardScaler` from Scikit-Learn API. The features are normalized independently on each feature by computing the relevant statistics on the samples in the **training** set. The StandardScaler is implemented with the following formula:

$$z = \frac{x - \mu}{s}$$

where μ is the mean of the training samples and s is the standard deviation of the training samples.

The same values of μ and s are then used to feature scale the test set data.

3.3 Train-Test Split

To perform experiments efficiently, we have to perform a train test split, so that we are able to test our models' accuracies and validation losses on our validation dataset, to see how well the models are able to generalize, to avoid the need to commit and submit our results to Kaggle for every experiments we make.

We performed **stratified** train test split using the `StratifiedShuffleSplit` from Scikit-Learn API to split our dataset into training set and validation set, with parameters as shown below.

```
# split train data into train and validation
sss = StratifiedShuffleSplit(test_size=0.1, random_state=123)
for train_index, valid_index in sss.split(scaled_train, labels):
    X_train, X_valid = scaled_train[train_index], scaled_train[valid_index]
    y_train, y_valid = labels[train_index], labels[valid_index]
```

Parameter `random_state=123` is used for reproducibility of our results and `test_size=0.1` is used for the split ratio of the data set.

We chose **stratified** train test split to split our dataset as our dataset consist of 990 samples, with 99 unique categories of species. As observed above, there is an equal proportion of data samples for each category (i.e. 10 samples for each of the 99 categories). Splitting our dataset with `test_size=0.1` gives us 9 samples of each category to train our models and 1 sample of each category to validate our model with. As our training sample for each category is very small, we believe that our choice of parameter, `test_size=0.1`, is optimal.

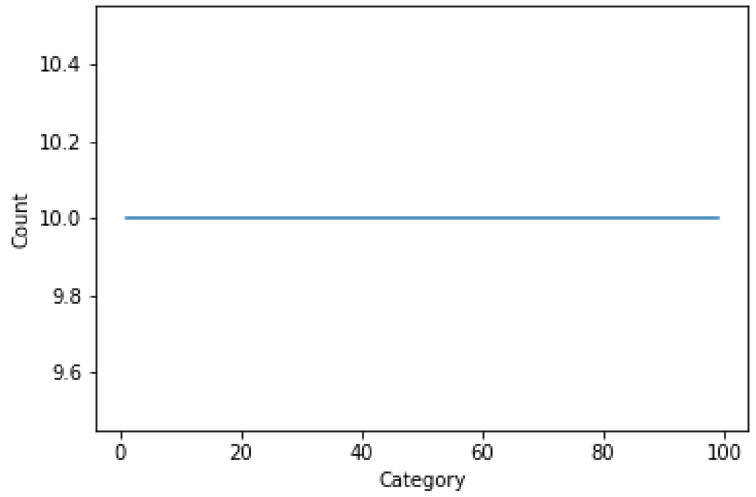


Figure 1: distribution

4 Models

4.1 Multi-Layered Perceptron

A Multilayer Perceptron (MLP) is a class of feed-forward Artificial Neural Network (ANN).

Modern computing systems outperform humans in the domain of numeric computations and manipulations. However, when it comes to complex perceptual problems, the human mind can effortlessly solve them at a speed that computing systems are unable to catch up with. Image recognition and classification, in particular, is one of such problems. Artificial Neural Network models take their inspiration from the brain.

In terms of structure, ANN is composed of a large collection of basic processing units called *neurons*, which are interconnected in some patterns to allow communication between the neurons. For a feed-forward ANN, each neuron in one layer has directed connections to the neurons in the next layer.

Perceptron

The perceptron (Roell 2017), or single-layer neural network, is developed in the 1950s and 1960s by the scientist Frank Rosenblatt (Nielsen 2015). It is the simplest model of neural computation. A perceptron takes several inputs, $x_1,\ x_2,\ \ldots,\ x_m$, processes them using a sum function and an activation function, and then produces a single binary output Y as illustrated in Figure 2.

Weights

To compute the sum function, Rosenblatt introduced weights, $w_1,\ w_2,\ \ldots,\ w_m$, associated with each directed connection link (Van Der Malsburg 1986). These weights are real numbers representing the importance of the respective inputs to the output. A *bias factor* theta, θ, is also defined as the threshold value for the sum function.

Activation Function

The activation function, also known as transfer function, is used to transform the output of the sum function to the perceptron's output. If the difference between the weighted sum and the bias factor, $\sum_{i=0} w_i x_i - \theta$, is positive, then the activation function should map it to 1, otherwise 0. An example of such function is the sign() function.

However, there are various other types of commonly used activation functions:

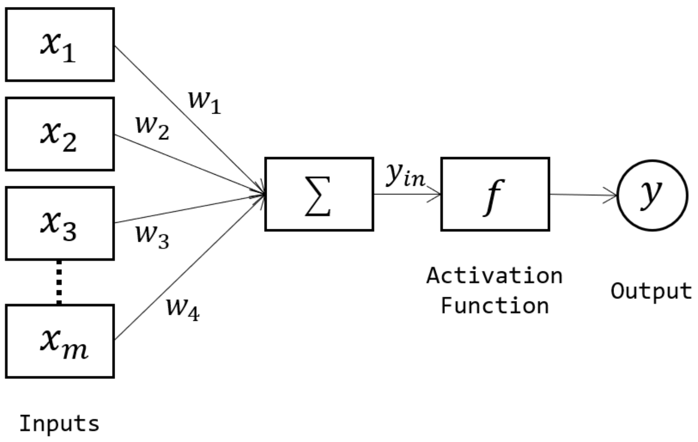


Figure 2: An illustration of Perceptron

- **ReLU Function:** ReLU stands for Rectified Linear Units. It is a type of activation function typically used in CNN. ReLU function takes in one value and returns that value it is positive or 0 otherwise. Mathematically, it is represented as follows:

$$\text{ReLU} = \max(0, x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

- **SoftMax Function:** This activation is normally used in a multi-classification problem. A SoftMax function is used for normalizing the weights into a probability distribution consisting of K probabilities. This transforms the weights to real numbers in the interval (0, 1), and helps indicate the dependency between classes, that is, the classes are mutually exclusive and exhaustive.

$$S_j = \frac{e^{a_j}}{\Sigma_{k=1}^{T} e^{a_k}}$$

Multi-layer Perceptron

As the sum function in perceptron model takes the linear combination of the inputs, a simple perceptron model is a linear classifier. One of the limitations of such a classifier is that it only works well for linearly separable classification problems. However, most of the times, without domain knowledge, we are unsure about whether the problem on hand is linearly separable.

To capture the more complex relationships in a classification problem, *hidden layers* are often added into the ANN so as to solve problems that are not linearly separable. This gives rise to the Multi-Layer Perceptron model invented by Minsky and Papert in 1969 (Minsky and Papert 1972).

A simple MLP consists of at least three layers: an input layer, an output layer and one or more hidden layers. An input instance X is fed to the input layer (including the bias factor), the weighted sum followed by the activation propagates in the forward direction, and the values of the hidden units Z_h are calculated (as shown in Figure 3). Each hidden unit is a perceptron by itself and applies a nonlinear activation function, such as sigmoid, to its weighted sum:

$$Z_h = \text{sigmoid}(w_h^T x)$$

This forward passing process continues in the same manner in all the hidden layers, with different weights in

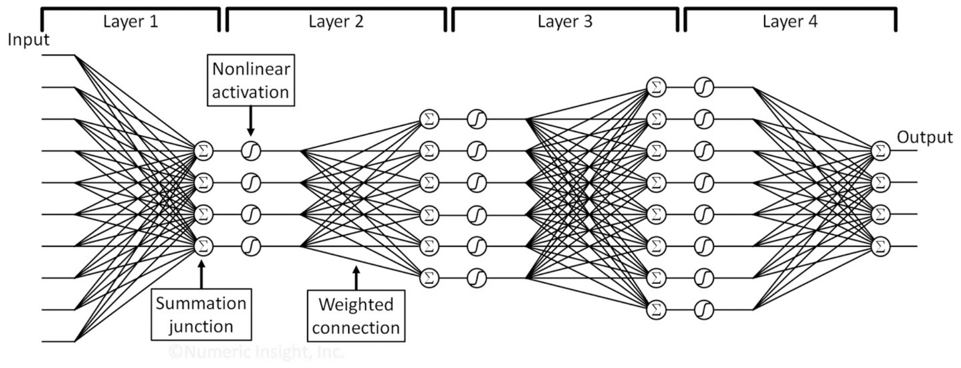


Figure 3: An example of a multi-layered neural network

each layer. The output layer then takes a linear combination of the values computed by the last hidden layer. For multiclass classification problem with K classes, there will be K output nodes, each computed from the corresponding weighted summation. A SoftMax function is added to normalize the weights into a probability distribution consisting of K probabilities. This helps indicate the dependency between classes.

Backpropagation Algorithm

Learning by MLP is essentially learning the weights associated to each link. To do so, we firstly initialize the weights to some values. After passing in each training instance, we try to adjust the weights such that the output of MLP is consistent with the class label of the training instance. A practical solution to update the weights is the backpropagation algorithm developed by D. E. Rumelhart, C. E. Hinton, and R. J. Williams (Rumelhart, Hinton, and Williams 1986).

In the *forward pass* phase of the algorithm, the information flows forward from the input layer through the hidden layers to the output layer. An output is computed based on the initial weights and the defined activation functions. This output is compared against the ground truth class label and a loss is computed based on a defined loss function.

In the *backpropagation* phase, the error is propagated back to the previous layer to update the weights between the two layers. This propagation continues until the first hidden layer. The actual update algorithm is based on gradient descent, where partial derivatives are used:

$$w_{t+1}{}^{(k)} = w_t^{(k)} - \lambda \frac{\partial E}{\partial w^{(k)}}$$

where k represents the layer, w_{t+1} represents the new weight, w_t represents the original weight, E represents the error function / loss function, and λ represents the learning rate.

4.1.1 Motivation

In our leaf classification problem, the given margin, shape, texture features are likely to have complex relationships which is not easily interpretable by humans. ANN, given its ability to capture complex relationships, is thus explored in our experiments.

Additionally, there is no evidence and it is not reasonable to assume that the species are linearly separable. As a result, we explored a simple MLP model instead of a simple perceptron model.

4.1.2 Implementation

A brief architecture of the MLP is as below:

```python
def create_model(weight_init='glorot_normal', lr=0.001, decay=0.0, drop=0.3):
    model = Sequential()
    model.add(Dense(512, input_dim=192, init=weight_init, activation='relu'))
    model.add(Dropout(drop))
    model.add(Dense(256, activation='sigmoid'))
    model.add(Dropout(drop))
    model.add(Dense(99, activation='softmax'))
    model.compile(loss='categorical_crossentropy',
                  optimizer=Adam(lr=lr, decay=decay),
                  metrics = ["accuracy"])
    return model
```

As there are 3 different types of features, i.e. margin, shape and texture, each represented by a 64-dimensional vector, the input layer consists of $64 \times 3 = 192$ nodes. This is followed by a hidden layer with 512 nodes and rectified linear unit (ReLU) as the activation function. After this, a dropout is applied before passing the signals to the next hidden layer, which consists of 256 nodes with sigmoid as the activation function. Again, a dropout is applied before reaching the output layer with SoftMax as the activation function. The output layer consists of 99 nodes, each representing one of the 99 species of leaves.

Here, we used categorical cross-entropy as the loss function because this is a multi-class classification problem. For the optimizer, Keras API offers a suite of different state-of-the-art optimization algorithms. However, typically we do not tune the optimization algorithm. Instead, we choose one approach beforehand and focus on tuning its parameter for the problem on hand. Here, we chose the `Adam` optimizer which was introduced by Diederik Kingma and Jimmy Ba in 2015. Adam is a well-recognized optimizer as it realizes the benefits of both Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp) (Jason Brownlee, 2017).

For the rest of the parameters, we conducted experiments to heuristically choose a better value for each parameter. The parameters we experimented with are summarized below:

Parameter	Description
`init`	Initializations define the way to set the initial random weights of Keras layers.
`lr`	Learning rate or step size. The proportion that weights are updated. Larger values (e.g. 0.3) result in faster initial learning before the rate is updated. Smaller values (e.g. 1.0E-5) result in slow learning right down during training.
`decay`	Learning rate decay over each update. Weight decay is applied when making the weight update, penalizing large weights.
`drop`	A regularization technique for neural network models proposed by Srivastava, et al. in their 2014 paper (Srivastava et al. 2014).
`batch_size`	Number of samples per gradient update.

4.1.3 Experimental Results

To find better values for the parameters, the simplest way is to conduct a controlled experiment. This means that, in order to investigate the effect of one parameter, we control the values of other parameters.

Alternatively, it is possible to use `GridSearchCV` from the Scikit-Learn API to test out the different combinations of parameters. However, this turned out to be too computational expensive for our devices and the Kaggle kernel to handle. Due to the lack of computational resources, we carried out our experiments using the controlled experiment approach.

When experimenting one parameter, we set the others to be at the default values listed below:

Parameter	`init`	`lr`	`decay`	`drop`	`batch_size`	`epochs`
Default value	glorot_normal	0.001	0.0	0.3	128	40

The experimental results are summarized below:

Parameter: `init`

Value	Log loss
uniform	0.130729
normal	0.131410
zero	4.708052
glorot_normal	0.136910
glorot_uniform	0.119318

From this experiment, we chose `init = glorot_uniform` as the random initialization algorithm, because it gives a lower log loss at epoch 40.

Parameter: `lr` and `decay`

Log loss		`lr`			
		0.1	0.01	0.005	0.001
`decay`	0.0	8.148133	0.008704	0.019991	0.135551
	0.0001	8.715399	0.011426	0.023289	0.123018
	0.000001	8.700432	0.041164	0.025569	0.129825

From this experiment, we chose learning rate `lr = 0.01` and `decay = 0.0` as this combination gives a lower log loss at epoch 40.

Parameter: `drop`

Value	Log loss
0.1	0.017277
0.2	0.022688
0.3	0.016495
0.4	0.024457

From this experiment, we chose the dropout rate `drop = 0.3` as it gives a lower log loss at epoch 40.

Parameter: `batch_size`

Value	Log loss	
	epoch 50	epoch 100
32	0.182798	0.162971
64	0.081175	0.075130
128	0.005962	0.003656
256	0.020365	0.014582

Value	Log loss	
512	0.012642	0.008039

Despite having the lowest log loss at epoch 40, a batch size of 128 yields an unstable log loss as the number of epochs increases up to 500 epochs (as shown in Figure 4). Hence, from this experiment, we chose the batch size `batch_size = 512` as it gives a stable and lower log loss.

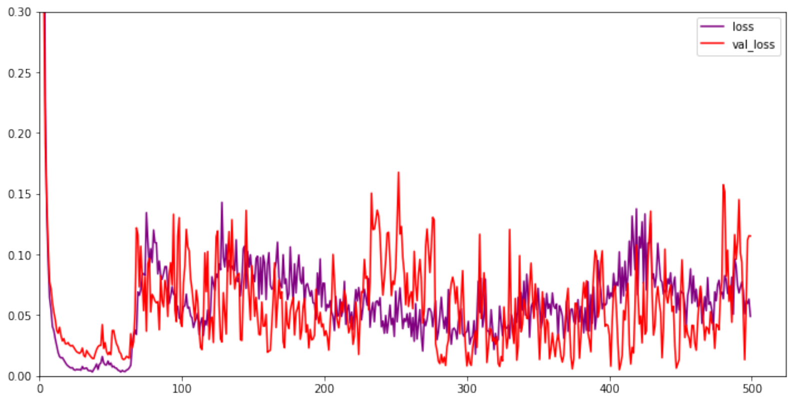


Figure 4: Log loss versus epoch for batch_size 128

In summary, the parameters are set to the following values in our final MLP model:

Parameter	`init`	`lr`	`decay`	`drop`	`batch_size`
Default value	glorot_normal	0.01	0.0	0.3	512

After tuning the parameters, we trained the MLP model until epoch 500 and plotted a graph to evaluate its performance.

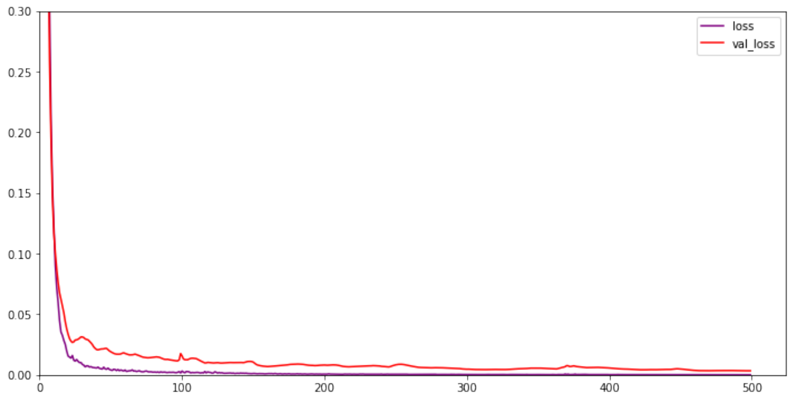


Figure 5: Log loss versus epoch for batch_size 512

From Figure 5, it is clear that the log loss is on a stable decreasing trend all the way until epoch 500. As a result, we deduced that the model does not have significant overfitting issue at epoch 500. Hence, the final MLP is developed with 500 epochs of training.

4.1.4 Evaluation

Due to the randomness in an MLP model (Brownlee 2017a), it is common to get slightly different results even with the same algorithm and the same parameter values. Generally, the above model performs quite well, with a log loss score ranging from 0.014 to 0.024. This score would put the submission at the top 18% in the competition leaderboard.

However, it is worth taking note that the tuning process described above is only based on heuristic trial and error. There is some possibility that the combination of parameter values we selected is not the optimal. Therefore, the MLP model might be able to achieve even higher accuracy and lower log loss if more detailed tuning is performed, which inevitably also requires more time and resources.

4.2 Convolutional Neural Network

In the following section, we are going to discuss the use of Convolutional Neural Network (CNN) in classifying leaves. As mentioned in the previous section, MLP are networks that are fully connected where each neuron in one layer is connected to all the neurons in the next layer. However, such trait makes MLP prone to the problem of overfitting. CNN addresses the issue through regularization whereby it assembles complex patterns in a set of given data by using a simpler representation.

CNN was originally inspired by the how cat and monkey sees things with the receptive fields in the visual cortex. Hubel and Wiesel then identified 2 visual cell types in the brain, namely the simple cells and complex cells (Hubel and Wiesel 1968). Based on this study, Hubel and Wiesel proposed a cascading model that can be used in pattern recognition (Hubel and Wiesel 1959). Inspired by the work of Hubel and Wiesel, Fukushima introduced the concept of 'neocognitron' which is where convolutional layers were first introduced (Fukushima 1980). After the initial stages of development of CNN, Yann LeCun proposed the use of back-propagation in CNN to identify handwritten characters and this becomes the foundation of modern research in computer vision (LeCun et al. 1989).

CNN gained widespread popularity in many fields of computer science such as computer vision where they are used to identify objects, classifying images, etc. which then leads to many applications such as self-driving cars, medical diagnoses, and in our case, plant leaf identification.

In a typical design of a CNN, there are a few commonly seen and used layers:

- **Convolutional Layer:** This is the cornerstone of any CNN. The purpose of a convolutional to reduce the size of the input without losing important high-level features hidden in the input. Typically, a **feature map** or **filter** of a much smaller dimension than the input slides through the input with a step know as **stride**. At every stride, a matrix multiplication is performed between the feature map and the portion of the input that the map is currently at. After performing the convolution operation, the dimensionality of the input is reduced. (However, sometimes the dimensionality remains or increases (Saha 2018)).
- **Pooling Layer:** To further reduce the computational power required, a pooling layer can be used to extract the most dominant feature of a certain portion of the data. Similar to the convolutional layer, a filter of a smaller dimension slides through the input and perform pooling operation at every stride. Typically, **max pooling** is used. This will return the maximum value among all the values under the current filter.
- **Dropout Layer:** To prevent the problem of overfitting, this regularization technique is performed. Dropout works by randomly removing neurons and connections between them from the neural network before training. By dropping out some of the connections, it 'forces' the neural network to learn more important features (Srivastava et al. 2014).
- **Dense Layer/Fully Connected Layer:** This is the layer where every node is fully connected to every node in the previous layer.

In LeCun's original paper in 1989, he provided a simple architecture of CNN, known as LeNet as shown in Fig. 6.

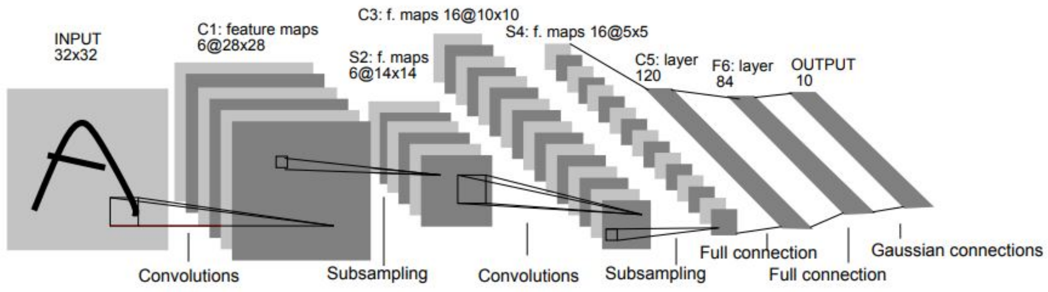


Figure 6: Architecture of LeNet

This then becomes a blueprint to the design of other CNN architectures in the future. Here is a short explanation to the LeNet Architecture:

1. First Layer: As illustrated in Figure 6, the input grayscale image is a handwritten character with a size of 32 pixels $\times$ 32 pixels. This input image is convolved by 6 filters or feature maps each of size 5×5 and a stride of one. This thus outputs a image of size $28\ (= 32 - 5 + 1) \times 28 \times 6$.
2. Second Layer: Each of these 6 convolved image is then pooled with an average pooling layer that has a filter size of 2×2 with stride of 2. This reduces the image dimension to $14\ (= 28/2) \times 14 \times 6$.
3. Third Layer: This is another convolutional layer with 16 feature maps each of size 5×5 and a stride of 1. Input is thus convolved to a dimension of $10\ (= 14 - 5 + 1) \times 10 \times 16$.
4. Fourth Layer: This is another average pooling layer that has a filter size of 2×2 with stride of 2. So the input is further reduced to $5\ (= 10/2) \times 5 \times 16$
5. Fifth and Sixth Layer: Fully connected layers each with 120 and 84 nodes respectively.
6. Output Layer: There are 10 outputs each represents a digit from 0 to 9 correspondingly.

4.2.1 2-Dimensional CNN

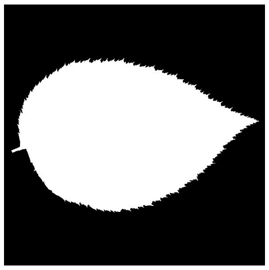
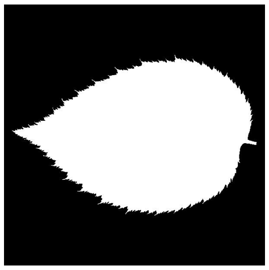
4.2.1.1 Motivation for 2-Dimensional CNN

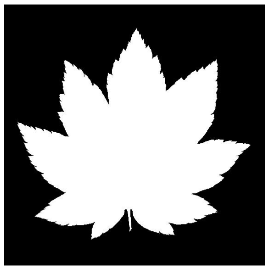
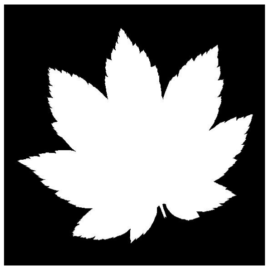
2-Dimensional CNN is well known for its application in the field of Computer Vision and they are frequently used in the ImageNet Competition, a Large Scale Visual Recognition Challenge. This is due to its ability to extract spatial information from images.

4.2.1.2 Implementing 2-Dimensional CNN

As explained above, due to the widespread popularity of CNN in classifying images, it is intuitive to start experimenting with a 2-Dimensional CNN for this Leaf Classification task, using the pre-processed images as input data.

As shown above, the pre-processed images are of different sizes and of grayscale image type. Grayscale images are made up of values between $0 - 255$, hence there is a need to normalize the images before feeding in to the CNN. Images are also resized to fixed dimensions, $(64 \times 64 \times 1)$.

Pre-Processed Image 805	Pre-Processed Image 1232
	
Alnus Maximowiczii	Alnus_Maximowiczii

Pre-Processed Image 164	Pre-Processed Image 962
	
Acer Circinatum	Acer Circinatum

As seen above, images of the same species are of different orientation (i.e. flipped and rotated). Data Augmentation is performed to increase the robustness of our CNN model, as it provides our model with more data and enabling our model to capture image invariances (Shorten 2018). To perform the normalization, resizing and data augmentation, `ImageDataGenerator` class from Keras API was used as shown below. Data augmentation is only used on the training data.

```
train_gen = ImageDataGenerator(rescale=1/.255,
                               width_shift_range=0.1,
                               height_shift_range=0.1,
                               rotation_range=0.1,
                               shear_range=0.1,
                               zoom_range=0.1,
                               horizontal_flip=True,
                               vertical_flip=True)
test_gen=ImageDataGenerator(rescale=1/.255)
```

A generic 2-Dimensional CNN was experimented with the architecture as shown below.

```
cnn_model = Sequential()
cnn_model.add(Conv2D(32, (5,5), padding='same', strides=(1,1),
              activation='relu', input_shape=(64,64,1)))
cnn_model.add(BatchNormalization())
cnn_model.add(Conv2D(32, (5,5), padding='same', strides=(1,1), activation='relu'))
cnn_model.add(BatchNormalization())
cnn_model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
cnn_model.add(Flatten())
cnn_model.add(Dense(256, activation='relu'))
cnn_model.add(Dropout(0.3))
cnn_model.add(Dense(99, activation='softmax'))
```

Two layers of 2-Dimensional Convolution Layer, each with 32 filters, filter matrix size 5×5 and activation function ReLU, were used to extract features of the images fed in. `BatchNormalization` layers were also added in between layers. `BatchNormalization` layer purpose is to normalize the activations of the previous layers at each batch and to speed up the learning of the CNN model. These are then connected to a `MaxPooling2D` layer and then to a fully connected layer (`Dense`) with 256 nodes. The fully connected layer is then linked to the output layer with 99 nodes matching 99 categories and a SoftMax activation function for multi-classification problem.

4.2.1.3 2-Dimensional CNN with Transfer Learning

Transfer Learning is method in which a machine learning model developed for another task is being reused as the starting point for a model on a second task (Brownlee 2017b). Weights from a pre-trained Neural Network in a task are used as a starting point for a machine learning model in another task, in this case Leaf Classification, and it helps the model learn faster and enhances its performance.

We experimented with **MobileNet** to build a 2-Dimensional CNN, as MobileNet is an efficient, light weight deep Neural Network which is based on a streamlined architecture, mainly for mobile and embedded vision applications (Howard et al. 2017). Due to this properties of MobileNet, it makes it the perfect choice to experiment on this Leaf Classification problem, as we will be able to train the MobileNet quickly and observe the initial accuracy and loss of it before further diving into other 2-Dimensional CNN architectures.

```
weights = "../input/mobilenet/mobilenet_1_0_224_tf_no_top.h5"

base_model = MobileNet(input_shape=((224,224,3)),
                       dropout=0.2,
                       include_top=False,
                       weights=weights,
                       pooling='max')
output_model = Dense(99, activation='softmax')(base_model.output)
model = Model(inputs=base_model.input, outputs=output_model)
```

Images were converted from grayscale to RGB, as MobileNet requires an input image of 3 channels, and resized the dimensions to $(224 \times 224 \times 3)$. The same Data Augmentation were performed with reasons justified above. The top layer of the MobileNet was replaced with one that fits this problem, a `Dense` layer as the output layer with 99 nodes and a SoftMax activation function for multi-classification problem.

4.2.1.4 Evaluation

Both Models are trained with parameters as shown below:

`optimizer`	`lr`	`decay`	`loss`	`batch_size`	`epochs`	`steps_per_epoch`
Adam	10^{-4}	10^{-6}	categorical_crossentropy	32	400	10

The results obtained are showed below:

Model	2-Dimensional CNN	MobileNet
Loss	1.272	2.667
Accuracy	0.596	0.747

Evaluating our 2-Dimensional CNN models in terms of model accuracy, MobileNet performs better than a generic 2-Dimensional CNN due to the fact that the MobileNet was trained with much more data and used to classify images in a new task. In terms of loss, a generic 2-Dimensional CNN performs better.

Comparing both CNNs with the other models that we experimented on, the stark difference in loss and accuracy led us to **not** proceed further with a 2-Dimensional CNN approach.

4.2.2 1-Dimensional CNN

4.2.2.1 Motivation

Two dimensional convolutional neural network (Conv2D) works great for classifying colored images that have red, green and blue color channels. However, only black and white images are available in our case, so the Conv2D model is not satisfactory. There are currently many researches about finding a representation of leaves. Two of the most popular approaches are content-based image retrieval using features on the images such as color and shape-based image retrieval. In fact, in the case of classifying leaf images, shape-based image retrieval provides more information about the leaves than features such as color. This is because most of the leaves are either green or brown (Caballero and Aranda 2010).

Given the inherent limitation of our data input, it is best that we turn to an alternative of Conv2D model, 1-Dimensional CNN. In fact, the data that we use contains a shape feature which is a contiguous descriptor of the contour of the leaf, and it a 1-dimensional representation of the 2-dimensional boundary of the leaf (Mallah, Cope, and Orwell 2013). Due to the sequential nature of the data, it also justifies our use of 1D CNN. In most case scenarios, recurrent neural network is used for modeling sequences of data and is commonly applied to audio signal analysis or machine translation. However, CNN is sometimes proven to be a superior alternative to modelling sequential data(Bai, Kolter, and Koltun 2018). Hence, we experimented with 1D CNN for leaf classification.

4.2.2.2 Implementing 1-Dimensional CNN

```
model = Sequential()
model.add(Conv1D(input_shape=(64, 3), filters=1024, kernel_size=6))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dropout(rate = 1 - 0.2))
model.add(Dense(2048, activation='relu'))
model.add(Dense(1024, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(99))
model.add(Activation('softmax'))
```

The above snippet of code shows the implementation of our 1-Dimensional CNN model. The data was first reshaped before feeding into the data, from a (990×192) dimension dataframe (excluding columns "id" and "species") to an array of $(990 \times 64 \times 3)$ dimension. The `Convolution1D` layer accepts input of dimensions $(batch\_size \times 64 \times 3)$, where our *batch_size* for this model is defined as 32. The reason behind our implementation of the `Convolution1D` layer as such is to input each category of column (i.e. *margin*, *shape*, *texture*) to a 1-Dimensional Convolution. The convolution layer also have parameters `nb_filter` which denotes the number of **filters** or **feature map**, and `filter_length` denotes the size of the filters, and an activation function ReLU. The output of the `Convolution1D` layer is then fed into a `Flatten` layer, which reshapes the output of the `Convolution1D` layer into an appropriate format to be fed in the `Dense` layer. Before being fed into the `Dense` layer, a `Dropout` layer was added in between with reasons as explained above. There are 3 `Dense` layer, with 2048, 1024 and 512 number of nodes. The output layer is a `Dense` layer with 99 nodes matching 99 categories and a SoftMax activation function for multi-classification problem.

4.2.2.3 Evaluation

After experimenting with parameters, we settled with a parameter as shown in the table below.

Parameter	`optimizer`	`lr`	`decay`	`loss`	`batch_size`	`epochs`
Value	Adam	10^{-4}	10^{-5}	`categorical_crossentropy`	32	100

After training for 100 epochs, 1D CNN is able to obtain the following result:

Loss	Accuracy	Validation Loss	Validation Accuracy
1.0887e−4	1	0.0096	1.000

Upon submission to Kaggle, however, we obtained a score of 0.03607. Nevertheless, this result is subjected to how randomness is implemented in Keras. Using the same model, it is able to achieve a score of 0.02605 (historical best score). Therefore we conclude that 1D CNN, despite not commonly being used for leaves classification, can give a very good accuracy when we feed it with sequential data such as shape as well as other features (i.e. margin and texture).

4.3 Multinomial Logistic Regression

In the following section, we are going to discuss the use of Multinomial Logistic Regression in classifying leaves. In order to fully appreciate the use of a multinomial logistic regression, we first must understand what Logistic Regression is.

Logistic regression takes many forms, in which the most basic form is known as a binary logistic regression. For the sake of simplicity, we are referring to binary logistic regression when logistic regression is mentioned. Logistic Regression is a classification model used to assign instances to 2 discrete classes. For example, it can be used to predict whether a student pass or fail a test or whether a customer will default in his bank loan. As the name suggests, logistic regression makes use of the logistic function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

To make sense of the above equation, we must start by explaining the concept of odds ratio. Odds ratio describes the ratio between the probability of a positive event occurring to the probability that it does not occur. As such, the more likely the positive event occurs, the larger the odds ratio.

$$\text{OddsRatio} = \frac{p}{1-p}, \text{where p is the probability of a positive event occurring.}$$

One limitation of odds ratio is that it ranges from 0 to $+\infty$. Hence, we apply log transformation to model the relationship between the target variable and our explanatory variables. By applying log transformation, we will get the log odds and the equation will take the following form:

$$\log(\frac{p}{1-p}) = b + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

The log odds range from negative infinity to $+\infty$. This is not useful since we are interested in the probability of an

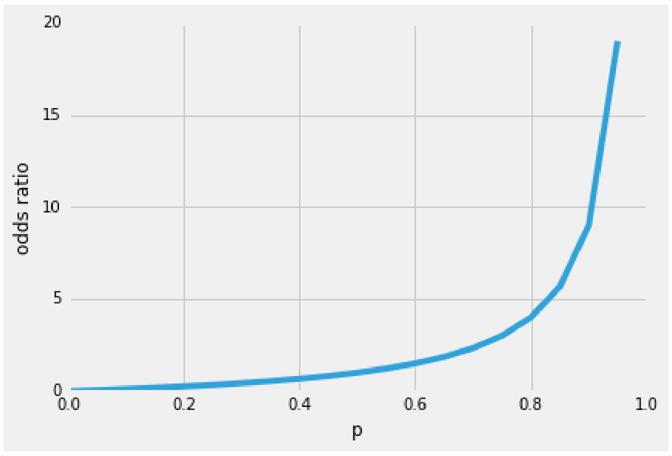


Figure 7: Odds Ratio

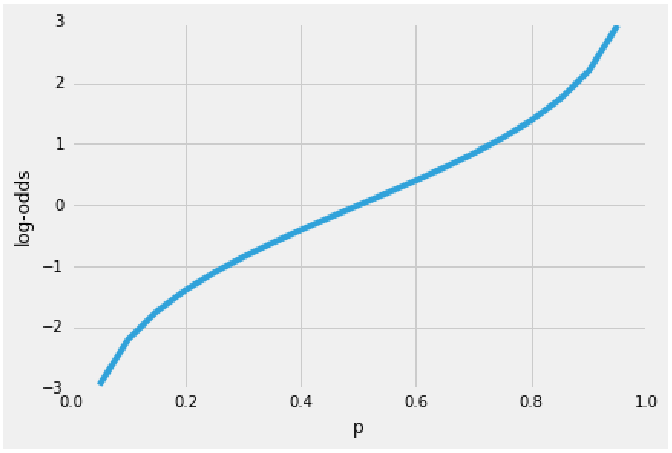


Figure 8: Log Odds

event occurring. We can take the exponent on both sides of the equation to arrive at the following equation:

$$\frac{p}{1-p} = \exp(b + w_1x_1 + w_2x_2 + \cdots + w_nx_n)$$

By re-arranging the equation, we will be able to obtain the probability of form:

$$p = \frac{e^z}{e^z+1} = \frac{1}{1+e^{-z}}, \text{ where } z = \exp(b + w_1x_1 + w_2x_2 + \cdots + w_nx_n)$$

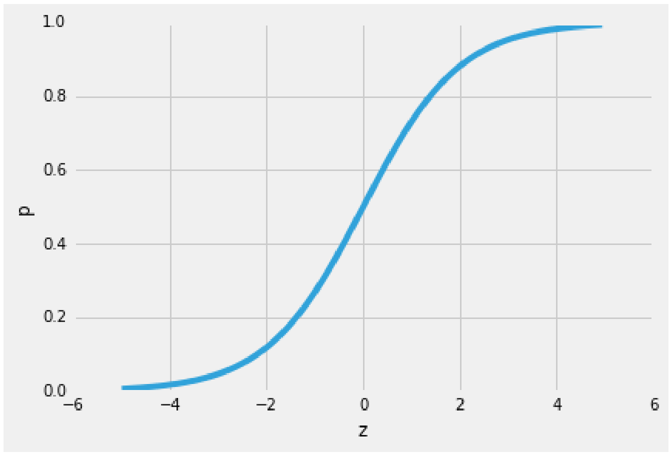


Figure 9: Logistic Function

With this manipulation, p will always range from 0 to 1. We also see that that p is in the form of the logistic function. Hence, logistic function is used for logistic regression. While the logistic function returns a probability score between 0 and 1, a threshold value should be selected to assign instances to classes. For a binary logistic regression, a threshold value is usually selected such that if :

$$p \geq 0.5, \text{ assign Class 1 and } p < 0.5, \text{ assign Class 0}$$

After discussing logistic regression, we can move on to explain multinomial logistic regression. Multinomial logistic regression essentially extends logistic regression. Just like the logistic regression, multinomial logistic regression is a classification method used to predict a nominal dependent variable based on multiple independent variables. The independent variables can be nominal or continuous in nature. While logistic regression will only result in 2 categories of outcome variable, multinomial logistic regression extends it as it is capable to predict more than 2 categories of outcome variable. Maximum likelihood estimation is used to evaluate the probability of categorical membership.

4.3.1 Motivation

In the dataset given to us, the dependent variable is categorical in nature while the independent variables are continuous in nature. There are 99 different species of plant and each instance can only belong to one species. We aim to correctly identify which category out of 99 categories an instance is referring to. Since we know that each instance can only belong to one species, the dependent variable has mutually exclusive categories. It is also exhaustive since we know that there are 99 species of plant. We also believe that the choice of the dependent variable is independent, i.e. choice of membership in one category is not related to the choice of membership in another category, and that there is no obvious multicollinearity issue. Hence, the assumptions for multinomial logistic regression are valid and we decided on using it for model building.

4.3.2 Implementation and Evaluation

```
#set a seed
seed = 42

#Logistic regression model
log_reg= LogisticRegression(penalty='l2',
                            dual=False, tol=0.0001, C=1.0,
                            fit_intercept=True, intercept_scaling=1, class_weight=None,
                            random_state=seed, solver='lbfgs', max_iter=100,
                            multi_class='multinomial', verbose=1,
                            warm_start=True, n_jobs= -1)

#Tuning the hyperparameters "C" which is the inverse of hyperparameter strength
#and "tol" which is tolerance for stopping criteria with GridSearchCV
classifier = GridSearchCV(log_reg,
                          param_grid = {'C':[0.1,0.5,1,10, 50, 100, 500, 1000, 2000],
                                        'tol': [0.001, 0.005,0.0001]},
                          scoring='neg_log_loss', refit='True', n_jobs=1, cv=10)

# Fit model.
classifier.fit(x_train,y_train)

# Make prediction for test data
y = classifier.predict(x_test)
y_prob = classifier.predict_proba(x_test)
```

In our multinomial logistic regression model, L2-penalty is selected to regularize the model and prevent overfitting. L2 regularization is selected over the L1 regularization since we believe that all the input features influence the output and hence there is no need force the weights to be zero as in the case of L1 regularization. In addition, since we have a large number of variables in the dataset, and that Limited-memory BFGS (L-BFGS) is well suited for optimization problem with a large number of variables, it is used as the optimization algorithm in our multinomial logistic regression model. To fine tune the hyperparameters `C` and `tol` of the multinomial logistic regression model, we have used a 10-fold cross validation **Grid Search**. By trying out each combination of hyperparameters specified in the grid, grid search allows us to find the best combination of hyperparameters that offers the smallest log loss.

After submission the generated csv file to Kaggle, we note that our multinomial logistic regression gives us a logloss of 0.03875.

4.4 Ensembled Model

Ensemble methods combine multiple models to improve performance. All ensemble methods share the following 2 steps:

1. Producing a distribution of base learners
2. Combining the base learners for prediction

Most ensemble methods use a single base learning algorithm to produce homogeneous base learners (e.g random forest using decision tree), but there are also some methods which use multiple learning algorithms to produce heterogeneous learners. Empirically, ensembles tend to yield better results when there is a diversity among the base learners.

4.4.1 Motivation

The approach of choosing only the 'best' model and discarding the others has a disadvantage: the network which had best performance on the validation set might not be the one with the best performance on new test data due to higher variance error. Ensemble enables further improvements by combining the decisions from multiple models to improve the overall performance.

This is especially true for neural networks. For example, if only a single MLP or CNN model discussed above is used, the results would have a relatively high variance. This is because neural networks are sensitive to initial conditions, including both the initial random weights and the statistical noise in the training dataset. Due to the stochastic nature of the algorithm, each time a MLP/ CNN model is trained for leaf classification, it may learn slightly (or dramatically) different mapping functions from inputs to outputs, leading to different performances. Besides, in neural networks, the variance increases and the bias decreases as the number of hidden units increase.

One way to decrease the variance error and improve the performance is through ensemble, in which the predictions from multiple good but different models are combined. This works when base models are independent, because different models will usually not make all the same errors on the test set (Brownlee 2018a). Ensemble thus leads to predictions that are less sensitive to the specifics of the training data and the choice of training scheme, leading to better and more stable predictions.

4.4.2 Implementation

Ensemble methods that combine predictions from different models can be varied and experimented by varying training data, models or the way that the predictions are combined. In our case, it is carried out by varying the ways to combine the results from models to arrive at the final prediction.

The simplest way to combine the predictions is to calculate the average of the predictions from the base models. However, some models may be more reliable and outperform the others, hence we would put more weights on their predictions. As such, averaging can be improved by weighting the predictions from each model, leading to a weighted average ensemble.

In order to learn how to best combine the predictions from each ensemble member, we experimented with various weights combination. Firstly, we selected 1D CNN , logistic regression and the MLP model discussed above. It is carried out by setting the weight on MLP's predictions to 0 first and then increase the weight gradually, with a change in 0.1 each time. The weights for the other base models are adjusted accordingly and the results are recorded. The following table shows some of the results worth highlighting:

CNN Weight	MLP Weight	Logistic Regression weight	Log Loss
0.0	0.0	1	0.03875
0.0	0.3	0.7	0.02053
0.0	1.0	0.0	0.01466
0.2	0.0	0.8	0.03675
0.2	0.7	0.1	0.01688
0.4	0.0	0.6	0.03644
0.4	0.6	0.0	0.01856
0.6	0.0	0.4	0.03686
0.6	0.4	0.0	0.02186
0.8	0.0	0.2	0.03782
0.8	0.2	0.0	0.02684
1.0	0.0	0.0	0.03927

4.4.3 Evaluation

As shown in the result table above, a single CNN, a single MLP and a single Logistic Regression model produce log loss of 0.03927, 0.01466 and 0.03875 respectively. A clear trend is observed from the results: as the weight on MLP increases, the log loss decreases and the performance gets better. The best performance in this case is achieved when only a single MLP model is used (log loss of 0.01466). This contradicts with the claim that ensemble helps decrease variance errors and improve performance. The first possible reason is that the MLP model included is optimized with hyperparameters and achieves much better result than the other two. Hence, it is highly unlikely that the other two models will add anything to the MLP model in an ensemble. This is proven to be a likely reason, because ensemble works when we choose the two models with similar log loss. Specifically, when only CNN and Logistic Regression models are ensembled (e.g. 0.2 on CNN and 0.8 on Logistic Regression), the resultant log loss 0.03675 is better than if either single model is used.

Besides, the models are likely to be dependent. This is because same data are used, and MLP and CNN both belong to neural network. Another possible reason is that the signal in the data may be explained by few very strong predictors. As such, all models would select and model similarly and thus ensemble does not help much in such cases. Hence, ensemble does not always improve performance.

However, the current ensemble still has much room for improvement. Although weighted averaging used is an improvement from averaging, this can be extended further. One possible future improvement is by implementing stacking. It is achieved by training an entirely new model to learn how to best combine the contributions from each ensemble member (Brownlee 2018b).

5 Results

After exploring the different models mentioned above, we came to the conclusion that the MLP would be our final model for submission. The main reasons are as follows:

1. It gives the best performance in terms of cross entropy (a.k.a. log loss) in our experiments for the leaf classification problem.
2. MLPs are suitable for classification prediction problems. They are very flexible and can be used generally to learn a mapping from inputs to outputs.

5.1 Evaluation Score and Ranked Position

For the Leaf Classification Competition, the evaluation is based on the multi-class logarithmic loss.

A set of predicted probabilities (one for every species) is to be submitted as the final result. The submission is then evaluated using the formula:

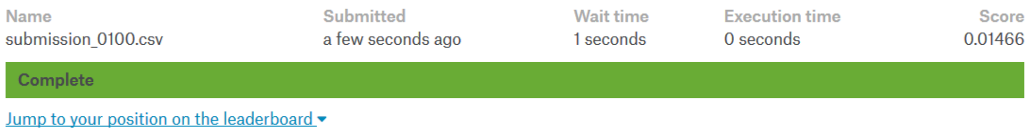
$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij})$$

where N is the number of images in the test set, M is the number of species labels, log is the natural logarithm, y_{ij} is 1 if observation i is in class j and 0 otherwise, and p_{ij} is the predicted probability that observation i belongs to class j.

In addition to prediction accuracy, log Loss takes into consideration the uncertainty of the prediction. More specifically, for each test instance, a higher confidence (i.e. prediction close to 1) for the ground-truth species results in a lower log loss. On the contrary, a low confidence for the ground-truth species is heavily penalized.

Based on this evaluation criterion, the performance of our final model is as follows:

Evaluation score	Ranked Position	Top
0.01466	151	9.45%

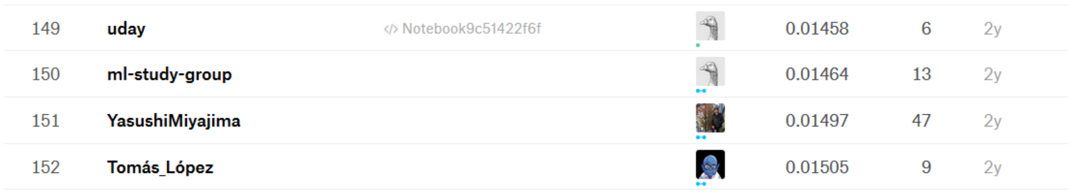


Name	Submitted	Wait time	Execution time	Score
submission_0100.csv	a few seconds ago	1 seconds	0 seconds	0.01466

Complete

Jump to your position on the leaderboard

Figure 10:



149	uday	Notebook9c51422f6f	0.01458	6	2y
150	ml-study-group		0.01464	13	2y
151	YasushiMiyajima		0.01497	47	2y
152	Tomás_López		0.01505	9	2y

Figure 11:

6 Conclusion

Image classification is a hot topic in recent years, with many tech giants, such as Google, investing huge amount of resources into this research area. As experimented earlier, there are multiple methodologies to tackle this problem, each having their own merits and disadvantages. Many meaningful insights can also be drawn from these experiments:

1. Feature engineering is an important discipline in terms of model building from image data. Even though deep learning algorithms have the ability to learn the features from the raw image data, this learning may not outperform the pre-processing by domain experts. For example, in this leaf classification problem, building deep learning models directly from the image inputs gives a significantly lower accuracy of 75%, whereas other models (e.g. MLP) using input data after feature engineering could easily achieve an accuracy of above 95%. However, it also needs to be noted that good domain knowledge is required for better extraction of features during feature engineering.

2. There is no one single best model that suits all problems. The performance of the different methods largely depends on the context, such as the input variables available.

3. In order to develop better models that suit the problem the best, it is important to read widely in terms of the relevant research findings. In this case, research findings for the leaf feature extraction and the applicability of the different machine learning algorithms helped us to devise our solutions and justify their performance.

References

Bai, Shaojie, J. Zico Kolter, and Vladlen Koltun. 2018. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling." *arXiv:1803.01271 [Cs]*, March. http://arxiv.org/abs/1803.01271.

Brownlee, Jason. 2017a. "How to Get Reproducible Results with Keras." *Machine Learning Mastery.* https://machinelearningmastery.com/reproducible-results-neural-networks-keras/.

———. 2017b. "A Gentle Introduction to Transfer Learning for Deep Learning." *Machine Learning Mastery.* https://machinelearningmastery.com/transfer-learning-for-deep-learning/.

———. 2018a. "Ensemble Learning Methods for Deep Learning Neural Networks." *Machine Learning Mastery.* https://machinelearningmastery.com/ensemble-methods-for-deep-learning-neural-networks/.

———. 2018b. "How to Develop a Stacking Ensemble for Deep Learning Neural Networks in Python with Keras." *Machine Learning Mastery.* https://machinelearningmastery.com/stacking-ensemble-for-deep-learning-neural-networks/.

Caballero, Carlos, and M. Carmen Aranda. 2010. "Plant Species Identification Using Leaf Image Retrieval." In *Proceedings of the Acm International Conference on Image and Video Retrieval - Civr '10*, 327. ACM Press. doi:10.1145/1816041.1816089.

Fukushima, Kunihiko. 1980. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position." *Biological Cybernetics* 36 (4): 193–202. doi:10.1007/BF00344251.

Howard, Andrew G, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. "Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *arXiv Preprint arXiv:1704.04861.*

Hubel, D. H., and T. N. Wiesel. 1959. "Receptive Fields of Single Neurones in the Cat's Striate Cortex." *The Journal of Physiology* 148 (3): 574–91. doi:10.1113/jphysiol.1959.sp006308.

———. 1968. "Receptive Fields and Functional Architecture of Monkey Striate Cortex." *The Journal of Physiology* 195 (1): 215–43. doi:10.1113/jphysiol.1968.sp008455.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. "Backpropagation Applied to Handwritten Zip Code Recognition." *Neural Computation* 1 (4): 541–51. doi:10.1162/neco.1989.1.4.541.

Mallah, Charles, James Cope, and James Orwell. 2013. "Plant Leaf Classification Using Probabilistic Integration of Shape, Texture and Margin Features." In *Computer Graphics and Imaging / 798: Signal Processing, Pattern Recognition and Applications.* ACTAPRESS. doi:10.2316/P.2013.798-098.

Minsky, Marvin, and Seymour A. Papert. 1972. *Perceptrons: An Introduction to Computational Geometry.* 2. print. with corr. Cambridge/Mass.: The MIT Press.

Nielsen, Michael A. 2015. "Neural Networks and Deep Learning." http://neuralnetworksanddeeplearning.com.

Roell, Jason. 2017. "From Fiction to Reality: A Beginner's Guide to Artificial Neural Networks." *Towards Data Science.* https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088): 533–36. doi:10.1038/323533a0.

Saha, Sumit. 2018. "A Comprehensive Guide to Convolutional Neural Networks — the Eli5 Way." *Towards Data Science.* https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-

way-3bd2b1164a53.

Shorten, Connor. 2018. "Data Augmentation on Images." *Towards Data Science*. https://towardsdatascience.com/data-augmentation-and-images-7aca9bd0dbe8.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15: 1929–58. http://jmlr.org/papers/v15/srivastava14a.html.

Van Der Malsburg, C. 1986. "Frank Rosenblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms." In *Brain Theory*, edited by Günther Palm and Ad Aertsen, 245–48. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-70911-1_20.