Udacity Machine Learning Engineer Nanodegree

Capstone Project: Kaggle Leaf Classification using TensorFlow

Author: Ke Zhang

Submission Date: 2017-05-23 (Revision 1)

Definition

Project Overview

Even today no one knows for sure how many plant species currently exist on earth. It is estimated however that the total number of plants is of the order of nearly half a million. That's why historically the classification of plant species has been very problematic and often resulted in duplicated identifications. Like the initiator of the Kaggle leaf classification competition[1] states, the automated plant recognition might have many important applications, including:

- Species population tracking and preservation
- Plant-based medical research or
- Crop and food supply management

The leaf dataset is a collection of 1584 labeled leaf images divided into 99 species which was provided by the Kaggle leaf classification competition ran from August 2016 to February 2017. It claims a total size of about 45Mb. 16 samples are available for each of the 99 species. For each sample, a vector of 64 margin features, 64 shape features and 64 texture features are already pre-extracted from the data provider. Both the pre-extracted features and the image data are used as input source in the model development of this project.

The binary leaf images and pre-extracted features originate from the leaf images collected by James Cope, Thibaut Beghin, Paolo Remagnino, & Sarah Barman of the Royal Botanic Gardens, Kew, UK and is hosted in the UCI machine learning repository[1].

Problem Statement

The plant classification using leaf samples is still a challenging and important problem today. The initiator of the Kaggle leaf competition has provided a set of pre-extracted features and leaf images as input and is looking for an automated way to identify the 99 species of plants described within the dataset using machine learning techniques.

A well-known machine learning technique in the area of image recognition is the convolutional neural network which has shown a great impact in many other machine learning projects and was able to make highly accurate predictions on datasets like the MNIST database of handwritten images[2].

The speciality of this project is that it uses multiple input sources with the pre-extracted shape information, margin and texture features provided along with the image data. The strategy of this project is to build a multi-input multi-output neural network combining the convolutional neural networks (1-dimensional for shape information and 2-dimensional for spatial binary images) with the pre-extracted features using fully-connected network[6]. That was necessary to achieve the required benchmark metric and to recognize the all samples in the testing set with 100% accuracy.

Metrics

This project uses a multiclass version of the *logarithmic loss* and accuracy as the evaluation metrics.

The *log loss* metric was suggested by the competition owner [4] and is defined as follows:

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

where *N* is the number of observations, *M* is the number of class 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* is in class *j*.

The *log loss* metric measures the recognition accuracy of the model on the test images in the leaf competition dataset. It can be understood as a soft measurement of accuracy that incorporates the idea of *probabilistic confidence* or the *cross entropy*[11] between the distribution of the *true labels* and the *predictions*. Thus, *log loss* is a measure to gauge the extra noise that comes from using a predictor as opposed to the true labels. By minimizing the *log loss* function, the model maximizes the *accuracy* of its classifier.

This leads to another commonly known *accuracy* metric used by this project which calculates the percentage of correctly recognized samples of the training set. The *accuracy* measure is easier to interpret by definition: it increases when the training goes well and gets 100% when the model predict all images correctly.

$$accuracy\ =\ \frac{\sum TP+\sum TN}{\sum total\ population}$$

The explanations above showed clearly that the *log loss* and *accuracy* metrics are appropriate and justified to be used for the leaf classification problem.

Analysis

Data Exploration

The leaf competition dataset contains in total 1584 leaf samples. The leaf samples are separated in two CSV files, one containing the labeled training set and the other one containing a “secret” testing set without labels. A quick summary is provided in the following table.

	Training Set	**Testing Set**
Sample Size	990 (10 per specie)	594 (6 per specie)
IDs Column present	yes	yes
Species Column present	yes	no
Number of Species	99	99
Number of Features	190 (64 features per category)	
Feature Categories	margins, shapes, textures	
Number of Null values	0	0
Number of Outliers	0	0

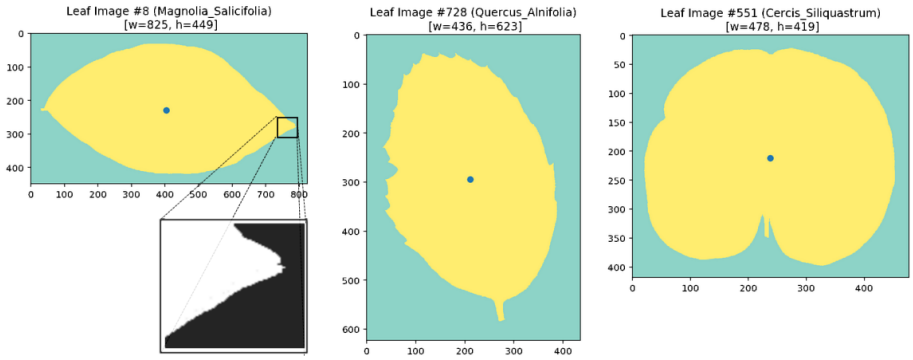
Along with the binary images the leaf data set provides three categories of pre-extracted features scaled to a range between 0 and 1. After first assessment no null or extreme values are found in the dataset. Here is how the training samples look like:

	id	species	margin1	...	margin64	shape1	...	shape64	texture1	...	texture64
987	1581	Alnus_Maximowiczii	0.001953	...	0.000000	0.007812	...	0.004883	0.000977	...	0.027344
988	1582	Quercus_Rubra	0.000000	...	0.046875	0.009766	...	0.000000	0.000000	...	0.083008
989	1584	Quercus_Afares	0.023438	...	0.031250	0.019531	...	0.000000	0.000000	...	0.000000

Exploratory Visualization

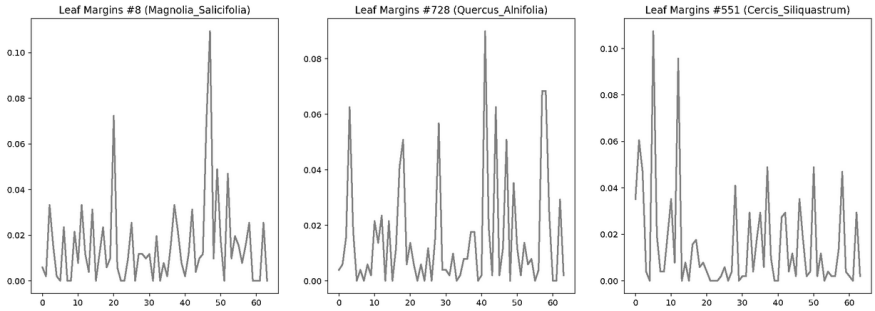
The leaf images are binary colored and have different dimensions. The resolution is highly enough to recognize the contour of the leaves. Visually the leaves of the same specie share similar image sizes, resolutions and orientations. Between the different leaf species it’s obvious

that they show distinctive characteristics like different shapes, contours or roughness at the edges.

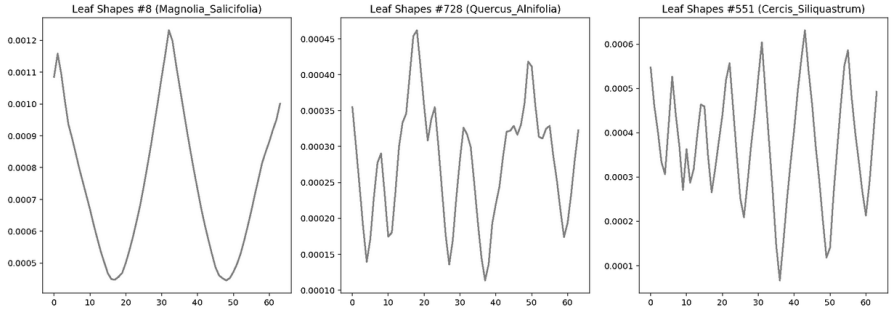


Next, the three categories of pre-extracted features are shown as graphs to get out some extra informations.

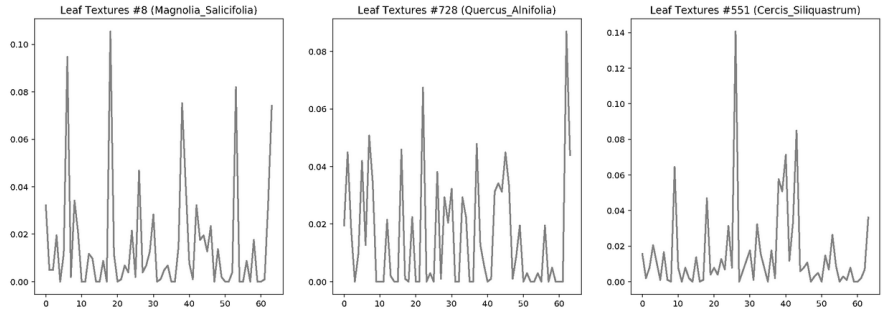
- The plot of the 64 margin features are unordered. No apparent shape can be recognized:



- The 64 shape features contain some unique patterns:



- Again, no apparent patterns can be identified in the 64 texture features:



After some explorations[17] it makes clear that the three feature categories are:

- *shapes*: a rotated, translated and normalized contour description of the leaf shape.
- *margins*: a measure of the roughness of the leaf margins. It's a filtered and normalized fine-scale margin histogram accumulation.
- *textures*: a normalized interior texture histogram accumulation.
- It is important to note that the provided leaf images are already preprocessed and don't have the interior textures anymore. The pre-extracted *textures* are of inestimable value since it could be the single source to the truth in some uncertain cases.

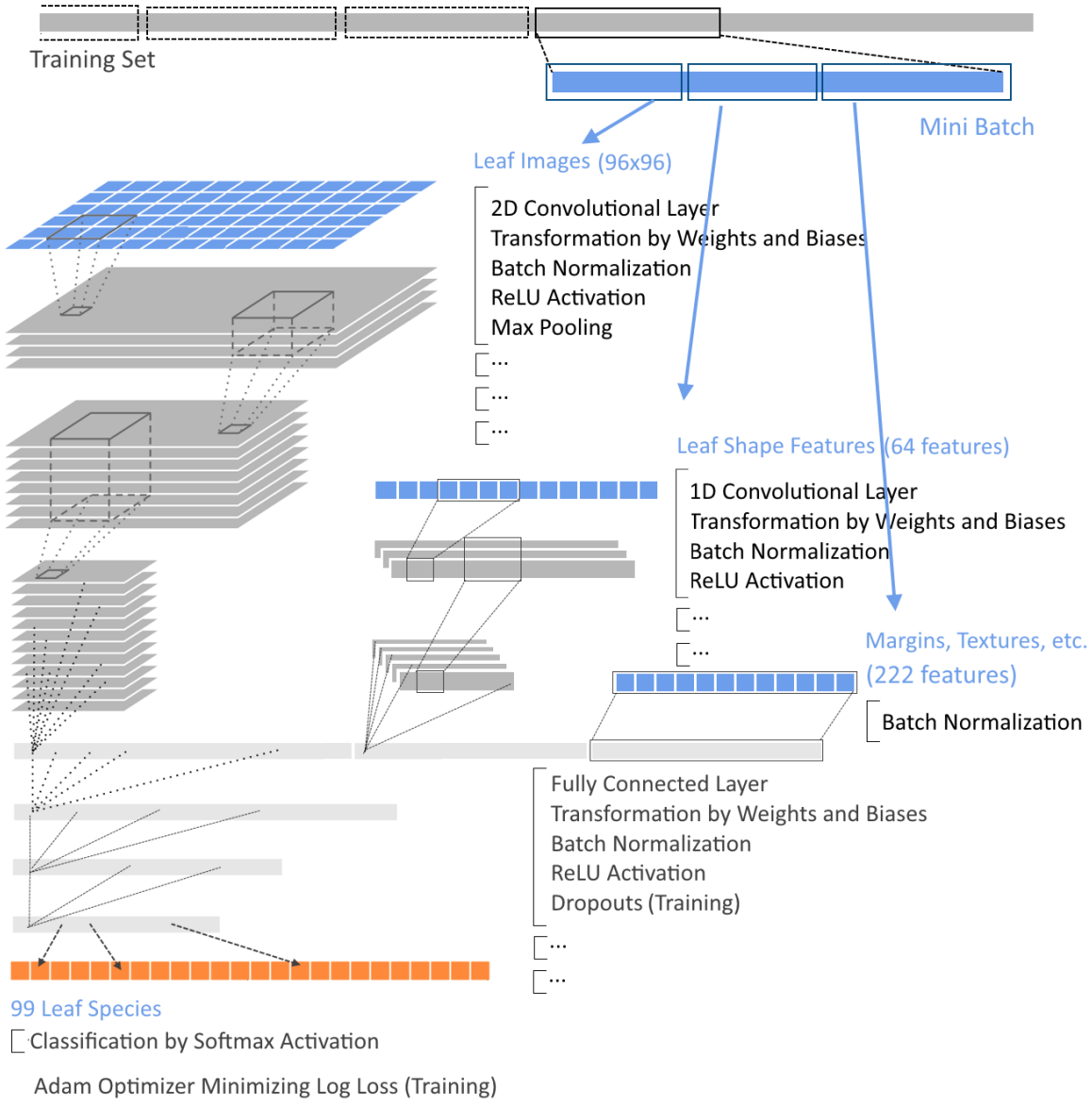
Algorithms and Techniques

This project will mainly follow the structure suggested by the Udacity Deep Learning course[5]. It will apply neural network techniques using the Google TensorFlow framework[8] to solve the leaf classification problem.

The solution is divided into three phases: feature engineering, preprocessing and model creation and training.

Image feature extraction techniques are applied in feature engineering where the additional features like centroid, enclosing ellipse and aspect ratios are extracted using *scikit-image* and *OpenCV* libraries. In the next phase, rescaled images and extracted features are prepared for the model training and saved to disk as checkpoint for quick reuse.

Once the input data is preprocessed, the model creates a neural network which takes the spatial leaf images, 1-dimensional shape data and pre-extracted features as inputs. Using a fully connected network, the model comprises them together into a multi-input multi-output computation graph[6]. Here is a simplified version of the complete neural network graph inspired by the MNIST TensorFlow deep learning tutorial[14]:



As shown in the model graph, the training set is first splitted into *mini-batches* to prevent the host machine from running out of memory. Initially the model divided the *990* training samples into mini-batches of *99* samples which then fed into the training loop every iteration. Once after *10* batches (*1 epoch* or *990* samples) the model is able to see all the samples.

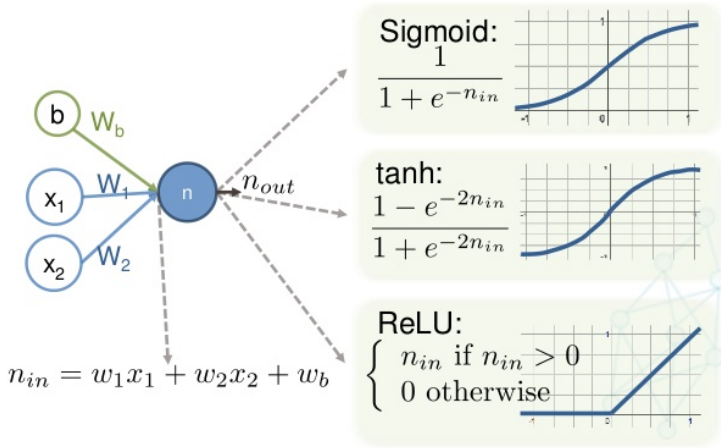
In the intermediate layers, all three input sources use the *batch normalization* provided by TensorFlow for whitening the feature values. This step is necessary to overcome problems with different scales, skewed or correlated data. In the default model settings, the *batch normalization* options are enabled to automatically center and rescale the input data before the activation function. In *batch normalization* the computing is done internally by calculating the statistics like the average and variance on each mini-batch while adapting the two learnable

scale *α* and offset *β* variables for each neuron to preserve the full expressiveness of the model. The small ϵ value is added to divisor to avoid division by zero. The definition is as followed:

$$BN(x) = \alpha \cdot \frac{x - avg(x)}{stddev(x) + \epsilon} + \beta$$

Then, each neuron does a weighted sum of all its inputs, added to a constant bias and feeds its result through the *ReLU* (Rectified Linear Unit) activation function. A activation function is necessary in neural networks in order to produce non-linear decision boundaries. The *ReLu* activation function, compared to the more classical *sigmoid* or *tanh* function, provides two major advantages[14]:

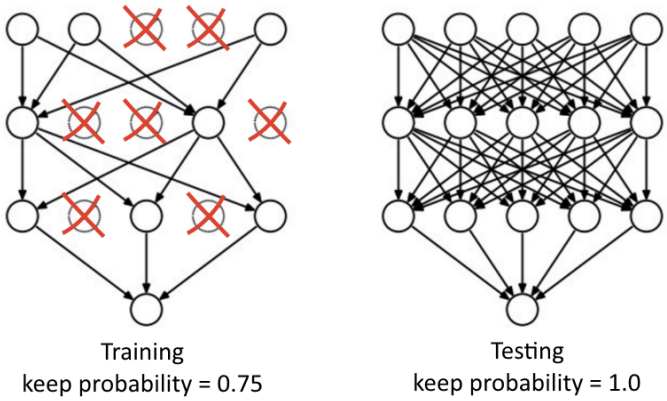
- It converges faster at the beginning and,
- it avoids problems when the network gets quite deep,
- … while the *sigmoid* and *tanh* functions tend to squash and vanish the values between 0 and 1 by adding more and more layers.



Additionally for the 2-dimensional spatial image data and 1-dimensional shape data, the model utilizes the convolutional neural network which is specialized to respect and extract useful image pattern information. In a convolutional network, each new neuron does a weighted sum of the pixel region (patch) above it, and then like the regular neurons the result is added by a bias and fed through the activation function. A convolution is the new layer by sliding the patch across the image in both directions. Pooling layer is subsequently inserted to gradually reduce the spatial size of input feature maps at the end of each 2D convolutional network. In this project the commonly adapted maximum pooling method is used.

After processing the two convolutional networks, the results of them are combined with the pre-extracted features from the mini-batch forming together as multi-input for the *fully-connected layer.*

During training the model uses the *dropout* technique for regularization of the *fully-connected network*. The *dropout* technique is popular because it is computationally cheap and effective in practice. It randomly resets a subset of the weights to prevent overfitting during training[8]:



A *keep probability* parameter is used in *dropout* technique to turn off (1-*keep probability*) of neurons. When the trained model is used for testing later, it always skips the *dropout* filter to get the stable result.

And eventually after performing the fully-connected network, the model applies the *softmax* regressions as output layer to terminate the neural network. *Softmax* is used as the termination function since it works well for a classification problem like this one and has the nice characteristic that its result clearly shows the winner element and at the same time it retains the original relative order of the input values. The final prediction is based on the output of the 99 *softmax* output neurons to classify the competing interpretations of the 99 leaf species.

Furthermore, to drive the training, the project uses the *log loss* function as discussed in the metrics section. The *log loss*, among all loss function is well-known in solving classification problems and measures how badly the model is at producing correct predictions. When the *log loss* falls through the loop, it means that the system is learning and is getting better at recognizing leaf species. During the complete training the neural network tries to tune its hyperparameters like weights and biases to minimize this loss.

Benchmark

In the Kaggle leaf classification competition, all submissions including the predictions for all testing set were evaluated using the multiclass *log loss* function. The winner of this competition *Ivan Sosnovik*[3] has created a pipeline using a combination of feature selection and engineering techniques. His pipeline contains a first-level logistic regression and a random forest classifier as his secondary classifier for the “confusion” or uncertain classes. With this model he achieved a perfect *0.00000* score on the leaf classification dataset. To this date, the best known submission using the convolutional neural networks had a *log loss* score of *0.06399*[12].

This project attempts to recreate his performance on the leaf species prediction using a multi-input and multi-output neural network.

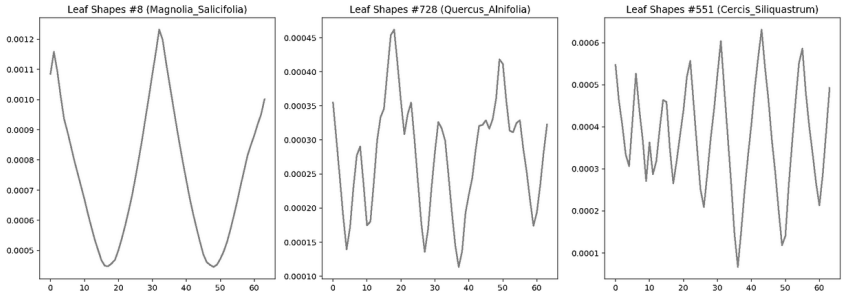
To be noted: the evaluation of the testing set predictions can be only obtained by submitting the predictions of the testing set to the Kaggle online site and the log loss score is calculated remotely.

Methodology

Data Preprocessing and Feature Selection

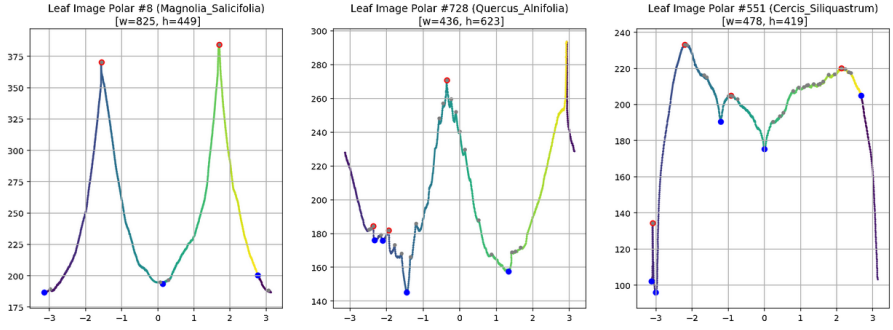
First of all, the sample size of 990 is quite small to be used to train a neural network. It is therefore necessary to extract additional information from the provided dataset.

From the discovery in the previous data exploration, it's clear that while margins and textures are histograms, the shape features have some continuous characteristics. This project will use it as input for the 1-dimensional convolutional neural network:

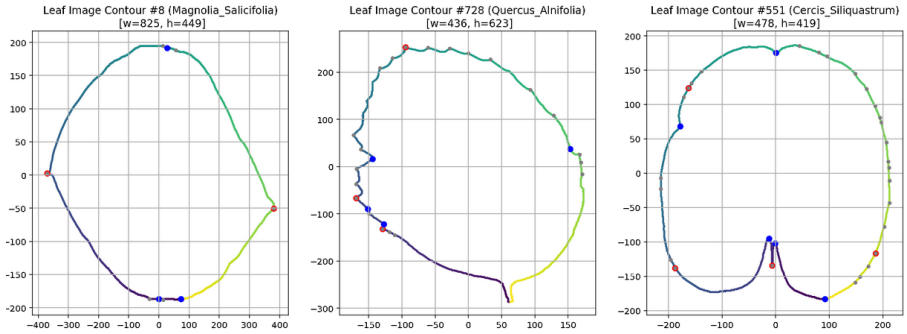


Next, I tried to collect additional image features using a image feature extraction method mainly described in a kernel notebook of another great competition participant[13]. The extraction process is as follows:

- The leaf contour is extracted from the image and is then turned into a time-series form using polar transformation. Further time-series analysis like finding the local extrema values are also performed here. The polarized graph below is enriched with extrema information and shows sometimes similar patterns as in the pre-extracted shape features .The red dots mark the local maxima, blue dots the local minimas and the smaller grey dots represent the "teeth" or the roughness of the leaf.



- Below is how it looks when the result of the time-series analysis is converted back to the cartesian coordinate system:

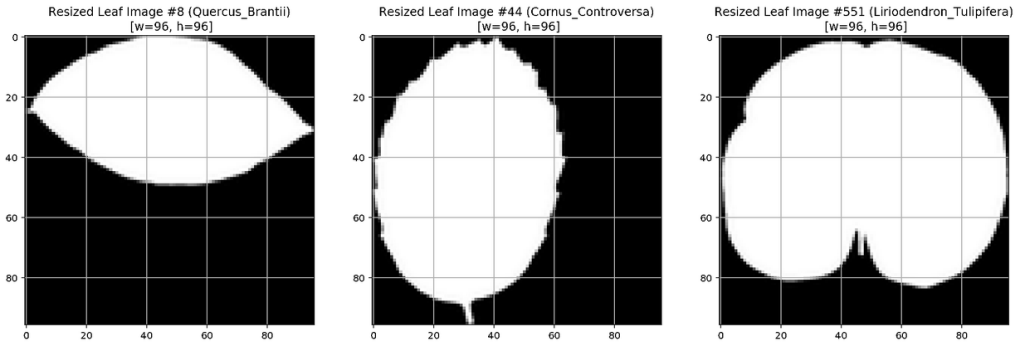


- Even more image features can be extracted using the OpenCV library[15]. Among others the aspect ratios, equivalent circle diameter, enclosing ellipse size, convexity and orientation of the leaf contour are used in this project:

	Sample #8	**Sample #728**	**Sample #551**
image width: **image height:** **size ratio:**	825 449 1.84	436 623 0.70	478 419 1.14
leaf area: **leaf perimeter:** **leaf area extent:** **leaf centroid:**	195445 1880 0.66 (404, 229)	130494 1680 0.67 (401, 466)	130485 1579 0.80 (372, 433)
margin maxima: **margin minima:** **number of teeth**	2 3 6	3 4 21	4 5 27
orientation: **convex:** **solidity:** **circle diameter:** **ellipse size:**	True False 0.99 499 (370, 693)	False False 0.95 408 (328, 517)	True False 0.96 408 (370, 458)

Furthermore in the preprocessing phase, all sample images are rescaled to a size of 96x96. As my hardware resources are limited, the rescaling of images to low-resolution is necessary to help to lower the computational costs for 2-dimensional convolutional neural network.

This size of 96x96 was carefully chosen to guarantee that the down-scaling don't vanish the differences between the images. When resizing, the leaf shape is cropped by its bounding box and then inserted into the black square-shaped background image to maximize the usage of the limited area while keeping the original ratio at the same time. Here are three rescaled images:



In summary, the project preprocessed the following features to prepare them as input for the neural network later:

- down-scaled sample images with binary color (96x96, 1 color channel)
- pre-extracted 1-dimensional shapes (64 points)
- pre-extracted margins and textures (64 features each)
- additional features extracted from the original images during the image preprocessing (30 image features):
    - image shape: width, height and size ratio
    - leaf shape: leaf area, leaf circumference and area/image ratio
    - leaf margin extrema: number of maxima, number of minima and number of teeth
    - OpenCV moments features: e.g. enclosing ellipse, equivalent circle diameter etc.

Normalization and scaling can be safely skipped in the preprocessing phase. The neural network later will apply batch normalization on all of the input features.

Implementation

There were many ways to construct a neural network. I started to create my model by adding the layers piece by piece. After some time the skeleton of my neural network is complete. Now it's time to tune all kind of parameters and number of layers. Below is a table listing the neural network settings of the initial solution:

	Leaf Images	**Leaf Shapes**	**Extracted Features**
Input Shape	(96, 96)	(64)	(222)
Convolutional Layers **- Kernel Size:** **- Strides:** **- Pooling Kernel Size:** **- Pooling Strides:**	2D CNN * 2 (8,8), (4,4) (3,3), (1,1) (4,4), (3,3) (6,6), (2,2)	1D CNN * 2 12, 6 4, 4 - -	-

Output Shape	(99, 36)	(99, 48)	(99, 222)
Fully Connected **- Input Shape:** **- Keep Probability:** **- Output Shape**	FNN * 3 (99, 306) 0.9 (99, 99)		
Loss Function	log loss		
Optimizer	AdamOptimizer with exponential decaying learning rates		

The most part of the algorithms and techniques shown in the table are already described as in the previous chapter. I'll discuss here about some of the difficulties and complicated functions during my implementation.

The first difficulty was that there were multiple possibilities to combine the input sources. I decided to use the leaf images rescaled to 96x96, the 64 shape data and all the 222 extracted features as the three input sources for my neural network. I came to this decision by experimenting the different input sources and using different settings separately. By making so, in each of the resulting neural network I got an accuracy over 80% for each of the input sources.

Another question was, to make the neural network how deep. The convolutional layers and the fully connected layers should be deep enough to make a training realistic on the small dimensioned laptop, but give accurate outputs at the same time. In the end, I had to keep the convolutional layers relatively "flat" with only *2* layers each and set the number of fully connected layers to *3*, since that's where all the input values are getting connected and things get interesting.

The *weights* for the matrix transformation are initialized randomly using the TensorFlow *truncated_normal* function that produces random values following the Gaussian normal distribution with a standard deviation of *±0.1*. To initialize the biases, I set them initially to small positive value like *0.1* as recommended so that neurons can operate in a non-zero range with the *ReLU* activation function.

```
W = tf.Variable(tf.truncated_normal([K, L] ,stddev=0.1))
B = tf.Variable(tf.ones([L])/10.0)
```

The *AdamOptimizer* is chosen to minimize the *log loss* function because it works well with high dimensional spaces like in this model here and it can safely pass the saddle points safely when lots of weights and biases variables are involved. Its initial learning rate starts fast with a value of $lr_{max}=0.001$ and decays with each step *i* exponentially to $lr_{min}=0.00001$:

$$learning\ rate(i) = lr_{min} + (lr_{max} - lr_{min}) \cdot e^{-i\ /\ decay\ speed}$$

Refinement

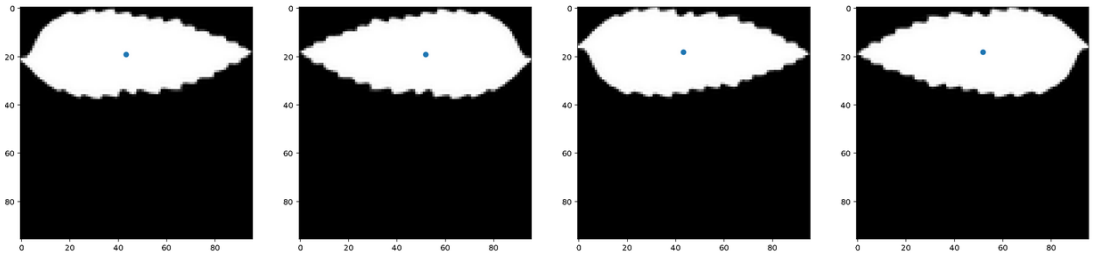
The model suffered from gradient vanishing problem during the early stages of the implementation caused by bad initialization. The model appeared to be not learning at all. It took quite a long time to check all the combination of the different initialization parameters and algorithms, until I finally figured out an appropriate solution for the leaf classifying problem. From all the choices between the hyperparameter spaces I took three snapshots during the settings engineering for the final model:

	Initial Selections	**Refined Choices**	**Final Choices**
Training Samples	990	990	**3960**
Weights Initialization	random normal initializer with zero mean and standard deviation between 0.0 and 2.0	random normal initializer with zero mean and standard deviation between 0.01 and 0.2	random normal initializer with zero mean and **0.1** standard deviation
Biases Initialization	constants between 0.01 and 1.0	constant 0.1	constant 0.1
Image Dimensions	32x32 to 128x128	64x64, 96x96	**96x96**
Dropout Keep Probability	keep probability between 0.5 to 1.0	keep probability between 0.9 to 1.0	keep probability of **0.9**
Learning Rate	initial rate between 0.001 and 0.2, target rate at 0.0001	initial rate at 0.02, target rate at 0.0001	initial rate at **0.001**, target rate at 0.00001
Batch Size	between 99 to 495 samples per batch	330 samples per batch	**396** samples per batch
Convolutional Layers, Kernel Sizes and Strides	1 to 5 convolutional layers with kernel sizes between 4 and 24 and strides between 1 to 4	2 to 3 convolutional layers with kernel sizes between 4 and 12 and strides between 1 to 4	2 convolutional layers with kernel sizes between 4 and 12 and strides between 1 to 3
Fully Connected Layers	2 to 5 fully connected layers	3 fully connected layers	3 fully connected layers
Number of Epochs	1000	10000	10000
Total Training Time	1 to 3 hours	2 to 8 hours	8 to 24 hours

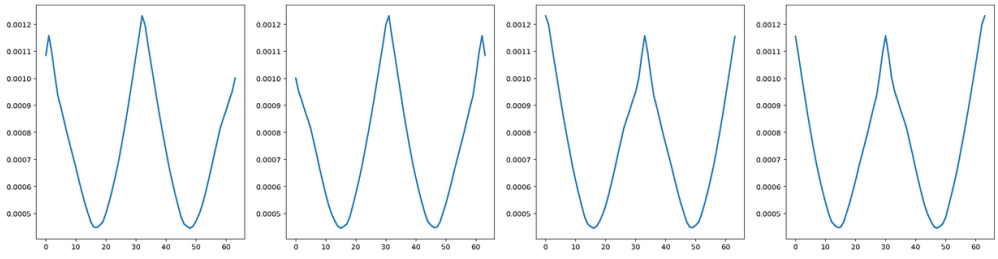
At the beginning of the training I started with the ideal settings giving the neural network as much degrees of freedom as possible. For example, the leaf images were resized to 128x128; both convolutional neural networks started with three layers each and the fully connected layers had even five layers to make a deeper neural network. But then I realized that within a limited training time and with limited computational resources it is impossible to finish the training. So I adjusted the settings dramatically and added the constraint of a maximum training time by trying to balance the number of layers in the neural network, the learning rate at the beginning and its decaying speed and the number of epochs.

After some refinements the model started to provide stable results with a constant *100%* accuracy, but its *log loss* score stucked at a range between *0.2* to *0.4*, which was far too high compared to the highscore board on the competition page. At this moment, some fine tuning was required. The first thing I did was to maximize the number of samples in one batch and randomized the training set after each epoch. Doing so, all of the components in the neural network like the gradients were better represented to the constraints, imposed by more samples and were therefore likely to converge faster towards the solution. The changes led to a *log loss* of *0.01021* on the testing set.

In order to improve the model performance even more, I analyzed the prediction output manually and realized that in some cases the leaf images have different positions than in the training set. E.g. some of the false identified leaf images have their stems on a different side than in the training images. It's obvious that the result could be enhanced by duplicating the training images by flip the image horizontally and vertically:



And I used the similar method on the shape features by reverse and mirror the shape data:



After all of the above-described, the number of training samples quadrupled from 990 to 3960 samples in total.

The last thing I optimized to approach the perfect zero *log loss* was to optimize the *dropout keep probability*. At some point during the training, I increased the *keep probability* to *1.0* so that the learning could start faster and skip to disturbing effects introduced by the *dropout*. But then I observed that while the *log loss* of the training set stops dropping, the *log loss* of the testing set bounced back up and started to disconnect from the training *log loss* score. After a few try and fail experiments with different dropout rates, I finally ended up with a *keep probability* of *0.9* which could bring back the quality of the testing set under control and at the same time didn't introduce too much noise during training so that the total training time was not too negatively affected.

In summary, the refinement process indicates that:

- Generating additional sample variants, e.g. by mutating the training images, can be beneficial in identifying the testing set.
- Adding layers in convolutional and fully-connected neural networks can increase the learning power of the model, however, the computational time will also increase exponentially.
- Increasing the number of samples per mini-batch and the total number of epochs can increase the total model performance.
- Smaller dropout keep probability rate helps to regularize the overfitting problem, but can also introduce additional noises into your system.

After all, the step by step refinements bumped the final *log loss* score from *0.04116* at beginning all the way down to the long desired goal of *0.00000*.

Results

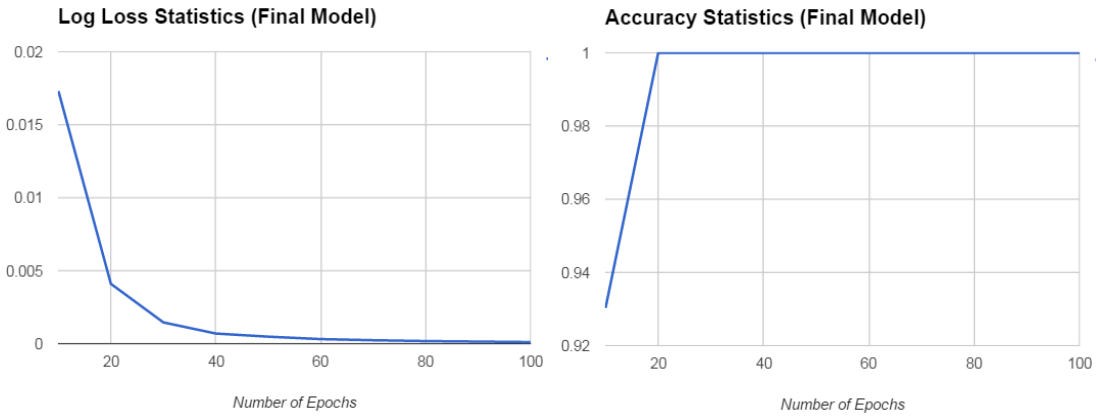
Model Evaluation and Validation

In the table below, three most representative model constellations are picked out listing their different set of hyperparameters and their evaluation metrics.

	Model 1	**Model 2**	**Model 3**
Training Samples	990	990	**3960**
Initial Learning Rates	0.01	**0.001**	0.001
Keep Probability	0.75	**0.95**	**0.9**
Samples per Batch	99	**330**	**396**
Number of Epochs	3000	10000	**10000**
Training Duration	~3 hours	~8 hours	~24 hours
Log Loss (Training)	0.00045	0.00000	0.00000
Accuracy (Training)	100%	100%	100%
Log Loss (Testing)	**0.04116**	**0.01021**	**0.00000**

Among the three neural network models, the last model produced the best *log loss* on the testing set with a score of *0.00000*. It derived from the initial solution mainly by lowering the *initial learning rate*, increasing the *dropout keep probability*, increasing the number of samples per batch and the number of training epochs.

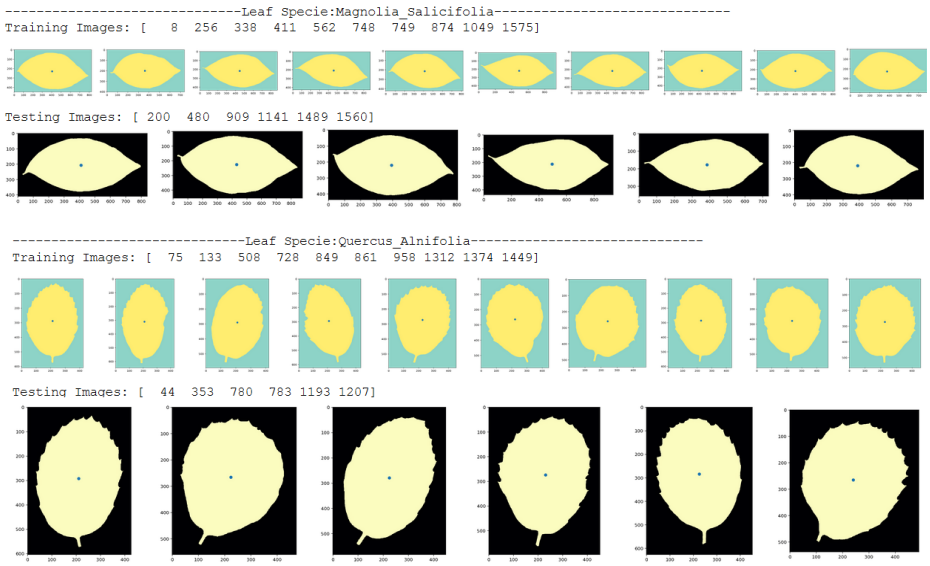
To evaluate the model performance, the system used 10% of the stratified training set for validation purposes. After every 10 training epochs it printed out the *log loss* and *accuracy* statistics of the current training batch and the validation set. The graphs below shows the *log loss* and *accuracy scores* on the validation set of the final model during training:

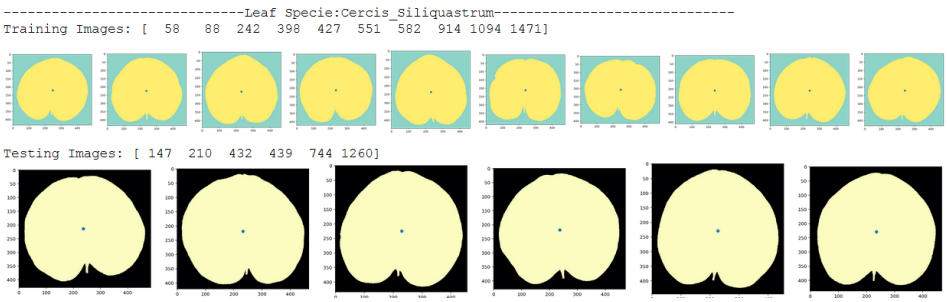


The *log loss* score dropped sharply from 0.02 to 0.0 within about 100 training epochs. The *accuracy* on the validation set high rocketed even faster and reached the 1.0 maximum *accuracy* in less than 20 epochs. By looking at the discrepancies between the training and validation set. The gap between the training and the validation set emerged after 100 epochs of training and there was no apparent sign of overfitting. This suggests that the final model is robust.

The subsequent evaluation of the testing set is done on the Kaggle competition server. The *log loss* score is determined remotely from the submitted predictions. The result is trustworthy, since the final model has no knowledge about the testing set and above all, the testing set itself doesn't even provide any label information. Thus, the 0.00000 *log loss* score suggests that the final model is in the position to generalize well to unseen data.

To finish, some of the predicted testing images are illustrated below. The first row with green background color shows the 10 training set samples and the row below with black background are the testing set images predicted by the final model classified to have the same leaf specie as the training samples.

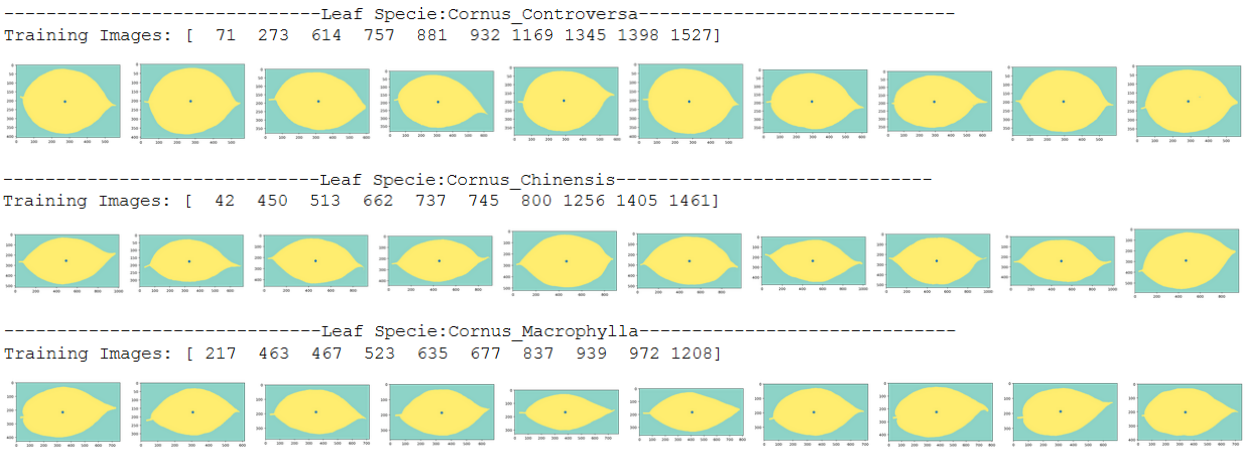




Justification

I must admit though that the predictions of the final model had a real *log loss* of *0.0039*. The problem here was that the predictions of final model were sometimes less certain about some leaf classes. The *softmax* output was not always 1.0. However, when using the *log loss* metric, by definition, only a perfect classifier would have a precisely zero. In other words, if all labels are predicted with absolute certainty. In a second submission to Kaggle, I applied an *argmax* function after the *softmax* on the end result, so that all predictions were rounded to 1.0 and I got finally a log loss of 0.00000.

In my opinion, the final model is an appropriate leaf classifier for real life applications, even with a log loss of *0.0039*. The majority of the test samples had certain predictions. Only for some classes, there were samples with uncertainties:



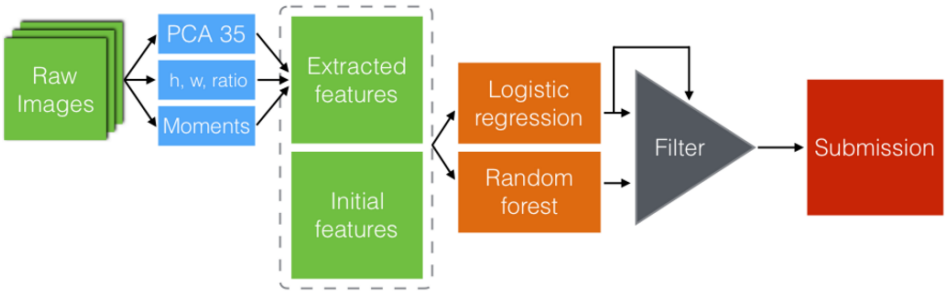
As you can see. The leaves above are from three different leaf classes, which are sub-species of Cornus. At first glance, the differences between the images are negligibly small. E.g. the images with id *1345*, *1461* and *217* look very similar. Thus, even for humans, there are cases where 100% certainty cannot be guarantied and subsequently the final model with a *log loss* of *0.0039* and *100% accuracy* should also be acceptable.

Conclusion

Free-Form Visualization

After my part was done, I was interested how the other participants solved this interesting leaf classification problem and which algorithms they chose.

In the Kaggle interview, the winner of the competition *Ivan Sosnovik*[3] described that his classification pipeline uses logistic regression and random forest algorithm to identify the leaf objects:



His first try was a pipeline using an optimized logistic regression classifier. In that approach, the *log loss* reached a value of *0.00686*. Then he modified the pipeline by adding a random forest classifier for 15 “uncertain” classes in case the logistic regression returned uncertain predictions.

It’s good to see that the winner had the same problems to deal with. And since the leaf classification dataset is small, it’s tempting to go straight with some simple and efficient algorithms as he did which point out to be powerful as well.

Reflection

This project presented a classification approach based on convolutional and fully-connected neural networks for recognizing the different leaf species using leaf images and pre-extracted features. The final pipeline consists of three phases that are feature engineering, pre-processing and model creation and training.

I learned from this project, how important it is, to have enough training samples and to extract useful information in the preprocessing phase. Sometimes it is helpful to be intuitive and generate additional training samples by yourself. For example, I quadrupled the number of leaf samples by flipping the images and by reverting and mirroring the shape features. As result, the predictions were more stable and had an improved performance on the testing set.

Last but not least, the final model fit my expectations for the leaf classification problem. I’m quite confident that this solution would also work in real world applications.

Future Improvement

The training of the neural networks was quite time-consuming, especially when the convolutional layers were active. TensorFlow is a distributed computing framework. With its deferred execution model it is possible to create create an execution graph in memory and send the actual tasks to a big farm of worker nodes. Next time, when I have to use neural networks, I would use a computing cluster like the Google Cloud[10], it would free most limitations happened during my training and I would build neural network with even more layers to benefit from deeper insights hidden in the data.

When testing the final model, I had the idea to check it with unseen leaf images from the real world. Unfortunately I then realized that the model depends additionally on the margin, texture and shape features provided with the dataset. On the UCI website where the data originally come from, there were no information available on how to extract those features. On a next occasion, I would study the paper about it, complete the missing piece in the pipeline and apply the final solution to real-world leaf images.

References

1. Kaggle Leaf Classification Competition: The subject of this proposal is derived from one of the most popular Kaggle playground competitions with more than 1500 participants.
2. The MNIST Database of Handwritten Digits by Yann LeCun et al
3. Kaggle: Leaf Competition Winner Interview
4. Kaggle: Log Loss Definition
5. Udacity: Deep Learning Course
6. Multi-input and multi-output models in Keras
7. MNIST TensorFlow Deep Learning Tutorial
8. Quora: Log Loss Definition
9. JamesMcCaffrey: Why you should use cross entropy error
10. Google Cloud: Machine Learning Engine
11. JamesMcCaffrey: Log Loss and Cross Entropy
12. Alrojo: TensorFlow Lab4 Kaggle
13. Lorinc: Feature Extraction From Images
14. Chang: TensorFlow Deep Learning (Chinese)
15. OpenCV Contour Properties
16. Kaggle: Leaf Competition Winning Kernels
17. UCI: leaves dataset