Almond Classification Using Neural Networks with Hybrid Learning

*COS 711 Assignment 2

1st Damian Jordaan
Department of Computer Science
University of Pretoria
Pretoria, South Africa
u20473509@tuks.co.za

***Abstract*—This report presents a neural network-based approach to classifying almond types and compares different learning techniques. The dataset comprises 2803 samples with 12 distinct features derived from almond images. In this dataset, various futures have missing elements, and some calculated features are present. Various data preprocessing techniques were applied, including missing value imputation, feature transformation and feature scaling. We performed Hyperparameter optimization using Bayesian optimization and grid search. The model's performance is evaluated through the mean of its accuracy and f-score over multiple independent runs. Additionally, we compared traditional gradient-based training algorithms, such as resilient backpropagation (RProp), against other optimizers and implemented a hybrid learning approach that integrates gradient updates from multiple algorithms. Results demonstrated that hybrid learning did not perform better than the optimal optimizer for this problem.**

***Index Terms*—Almond classification, Neural networks (NN), Hybrid learning, Hyperparameter optimization, Resilient backpropagation (RProp)**

I. Introduction

This report focuses on the problem of almond classification, which involves classifying almonds into one of three types—Mamra, Sanora, or Regular—based on numeric features extracted from images.

A. Almond Dataset

The dataset used in this study includes 2803 samples. Due to constraints in the imaging setup, some features have missing values that need to be processed. Additionally, some of the features are calculated from other features. This report will detail some of the different data preprocessing methods, discuss their performance, and attempt to find the method that should yield the best performance.

B. Hyperparameter optimization

Hyperparameter optimization is crucial to improving a model's predictive performance. This optimization process involves tuning parameters such as learning rate, batch size, and network architecture. We use the Bayesian optimization strategy to identify the best-performing configurations based on general parameter tuning and then further tune the parameters using a grid search strategy.

C. Neural Network gradient-based optimizers and Hybrid Learning

This study also explores hybrid learning techniques by integrating the strengths of multiple gradient-based optimizers, such as resilient backpropagation (RProp) and Nadam. By averaging the weight update value from different algorithms, hybrid learning aims to enhance convergence speed and stability, potentially leading to improved accuracy over conventional training methods. This report details the hyperparameter tuning for various optimizers and compares their performance for this problem. Apart from RProp, we select the best-performing optimizer to create a hybrid learning optimizer that averages the weight updates from RProp and the best-performing optimizer.

D. Experiment Results and Model Performance

The model's performance is measured using a combination of accuracy and f-score. It is then given a value that is the minimum of the two metrics.

II. Background

This study approaches the classification task using a supervised neural network model trained on a dataset of numeric features extracted from almond images. The dataset includes 2803 samples with 12 features per sample, capturing characteristics like length, width, and thickness. However, due to image orientation limitations, only two of these three primary dimensions are present for each sample, which complicates the classification process. Careful preprocessing is necessary to address these missing values and scale features appropriately for effective neural network training.

A. Equations

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

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3}$$

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

B. Hyperparameters

General Hyperparameters

- **test_size**: Determines the proportion of the dataset reserved for testing the model's performance after training. Commonly expressed as a decimal (e.g., 0.2 for 20% of data), this split helps evaluate the model's generalization ability on unseen data.
- **batch_size**: Specifies the number of samples the model processes before updating the model's parameters. Smaller batch sizes allow for more frequent updates, potentially leading to faster convergence, while larger batch sizes provide more stable updates but may slow down learning.
- **epochs**: Defines the number of complete passes through the entire training dataset. Higher values allow the model more time to learn, but excessive epochs may lead to overfitting, where the model performs well on training data but poorly on test data.
- **patience**: Used in early stopping criteria to prevent overfitting. Patience is the number of epochs to wait after a performance improvement before stopping training. If the model's performance (e.g., validation loss) does not improve within the specified patience interval, training will stop early.
- **min_delta**: Sets the minimum threshold for a performance improvement to be recognized during training. Changes smaller than min_delta are considered negligible. This value works with patience to help determine when training should stop if progress stalls.
- **starting_hidden_layer_size**: Specifies the number of neurons in the initial hidden layer. This value can influence the model's ability to capture complex patterns; too few neurons may limit learning, while too many can increase computational cost and risk of overfitting.
- **number_hidden_layers**: Indicates the total number of hidden layers in the neural network. Additional hidden layers enable the network to learn more complex representations and increase model complexity and computational requirements.
- **hidden_layer_activation**: The activation function is applied to the neurons in each hidden layer. Common options include ReLU, tanh, or sigmoid functions, each introducing non-linearity to help the network model complex patterns.
- **output_activation**: The activation function applied to the output layer shapes the network's output format. For instance, a softmax function is often used for multi-class classification, while the sigmoid function is typically used for binary classification.
- **learning_rate**: Controls the size of steps taken during parameter updates in training. A high learning rate can accelerate training but may lead to instability, while a low learning rate provides more precise updates but may slow down the learning process.
- **loss:** The loss function used by the network. Example are

Other Optimizer-specific Parameters

- **beta_1**: A parameter used in adaptive optimizers like Adam that controls the exponential decay rate for the first moment (mean) estimate of gradients. It typically has a value close to 1 (e.g., 0.9) and helps stabilize updates by balancing the influence of past gradients on current learning.
- **beta_2**: Another parameter in optimizers like Adam, it controls the exponential decay rate for the second moment (variance) estimate of gradients. Typically set close to 1 (e.g., 0.999), it helps manage noisy updates, especially in the presence of sparse gradients.
- **epsilon**: A small constant added to the denominator in optimization algorithms (e.g., Adam, RMSprop) to prevent division by zero. It ensures numerical stability during weight updates, particularly when gradient values are very small.
- **initial_step_size**: In gradient-based optimizers, this parameter defines the initial size of each update step. It acts as a starting point for step size adjustments, balancing early progress with the potential need for later fine-tuning.
- **increment_factor**: Used in algorithms that adjust step sizes dynamically, such as resilient backpropagation (RProp). When the gradient's direction remains consistent, this factor increases the step size, allowing the model to converge faster when progress is steady.
- **decrement_factor**: Also used in step size-adjustment algorithms, it reduces the step size when the gradient's direction changes (indicating possible oscillation). This helps the model avoid overshooting and promotes smoother convergence.
- **min_step_size**: Sets the minimum allowable step size in adaptive optimizers. By capping the smallest steps, it prevents the model from becoming too slow to learn, even if the gradient becomes very small.
- **max_step_size**: Defines the upper limit for step sizes in algorithms with adaptive step adjustments. This cap prevents overly large updates that could destabilize learning, especially if gradients are unexpectedly high.

C. Data Exploration Analysis

The data in the almond classification data set is evenly split between the three types of almonds. Additionally, it is also evenly, per almond type, split between the different measurement formats. This being the case unbalanced classes is not a concern, and is therefore, not accounted for.

D. Optimizers

- **RProp (Resilient Backpropagation) Definition**: RProp, or Resilient Backpropagation, is a gradient-based optimization algorithm that adjusts the step size based solely on the direction of the gradient rather than its magnitude. By only considering the sign of the gradient, RProp aims to avoid the problems of vanishing or exploding gradients, making it suitable for networks with oscillatory or unstable learning. It adapts step sizes dynamically: increasing them when gradients consistently point in the same direction and decreasing them when the gradient direction changes.

- **Adam (Adaptive Moment Estimation) Definition**: Adam is a widely used adaptive optimization algorithm that combines ideas from both Momentum and RMSProp. It maintains running averages of both the gradient (first moment) and the squared gradient (second moment) to adapt the learning rate for each parameter individually. This approach enables faster convergence by adjusting step sizes based on both the recent gradient history and its variance, which helps stabilize updates and improve performance in sparse or noisy data settings.
- **Nadam (Nesterov-accelerated Adaptive Moment Estimation) Definition**: Nadam extends Adam by incorporating Nesterov momentum, which applies a correction step that "looks ahead" before updating the parameters. This foresight helps to reduce oscillations, especially in the early stages of training. Nadam retains the benefits of Adam's adaptive learning rates and momentum while improving convergence speed, making it especially effective for models that benefit from accelerated gradients and adaptive learning adjustments.

Tensorflow's built-in classes were used by all other optimizers.

III. Experiment Setup

A. Data Preprocessing Variations

For this experiment, three different data preprocessing methods were tested. Additionally, all calculated features were removed. Moreover, all features were normalized - except for length, width, and depth, which were instead min-max normalized to be within '0 and 1.' Furthermore, the length, width, and depth features are processed as one large feature when it came to applying normalization. In other words, they share a min and max value, as well as a mean and standard deviation value.

- Method 1: In this method, the length, width, and depth were combined into length A and length B. Additionally, three columns are added - L, D, and W. In these columns, a '1' will be used where the original columns were present and a '0' will be used if no data was present in the original column.
- Method 2: Same as method 1, except instead of adding an 'L, D, and W' columns, one-hot encoding was used for the representation of the original columns. For example, if length and width were present, the representation would be 'L x W'
- Method 3: In this method, a '-1' was added wherever there were missing values in the length, width, and depth columns.

B. Neural Network Basic Architecture

For this experiment, a pyramid structure was used for the neural network, where an initial number of hidden layer neurons, as well as hidden layers were defined. The program rule then calculates the number of neurons in the subsequent hidden layers based on the number of hidden layers, such that the number of neurons in each hidden layer decreases evenly until the number of output neurons is reached.

A pyramid structure for a neural network has been shown to have good generalized performance. [1]

C. Exploration Testing Setup Data

This outlines the testing methodology for the three variations of data preprocessing. The three methods were tested on the basic pyramid neural network structure with a starting hidden layer size of '64-128', a test size of '0.2', and a batch size of '32' over 200 epochs. Thirty independent tests were conducted for each method. Additionally the "Adam" optimizer was used.

D. Exploration Testing Setup General Hyperparameters

For this experiment, all globally applicable hyperparameters, excluding epoch, were tuned using Bayesian optimization. This was done over 500 test cases.

E. Exploration Testing Setup RProp

In this test, only RProp's own hyperparameters were tuned, this being initial step size, increment factor, decrement factor, max step size, and min step size. The parameters were also tuned using Bayesian optimization over 500 test cases.

F. Exploration Testing Setup Optimizers

In this test, a variety of different optimizers were tested, each with their own hyperparameters being tuned individually. The optimizers that were tested are "adam", "sgd", "rmsprop", "adamax", "adagrad", "adadelta", "nadam", "ftrl", "adamw", "adfactor"

G. Final Data Preprocessing

Based on the results of the exploration testing, a modification was made to the data preprocessing whereby the measurement features were processed using standard deviation instead of min-max. The reason for doing this was to allow for more identity in the data, as processing with min-max can lead to the loss of identity in the data.

H. Final Testing Setup

Based on the results of the exploration testing, the best optimizer was found to be "Nadam." As such, the hybrid optimization method is a combination of RProp and Nadam. This is implemented as follows: First, the weight update of RProp is calculated, then the weight update of Nadam is calculated, then a weighted average is calculated for the two variables, and that weighted average is then applied to the neuron weight.

The hybrid optimization in addition to RProp and Nadam were all tested with the best performing hyperparameters from the exploration testing. Additionally, a grid search was performed for each using the hyperparameters batch size and hidden layer activation function. The batch size was tested for 3 possible sizes - 8, 32, and 64.

IV. Research Results

A. *Exploration Testing Results Data*

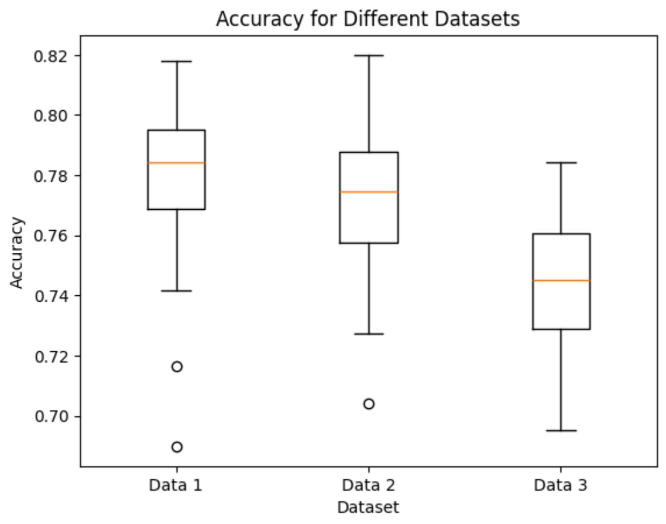

Fig. 1. The Accuracy Box plot for the different data preprocessing methodologies.

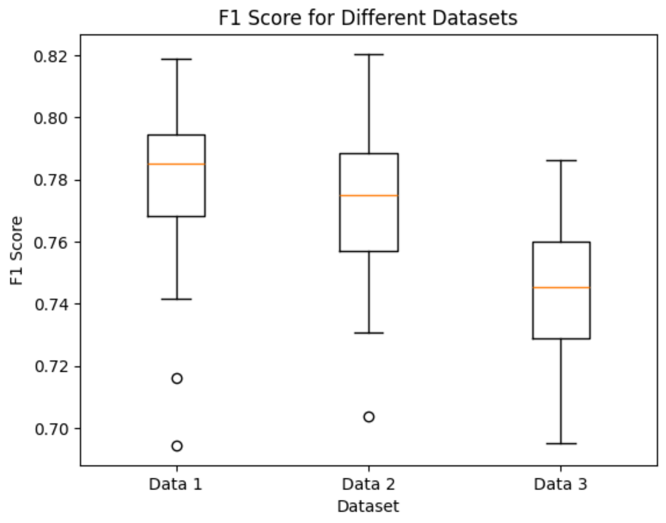

Fig. 2. The F-score Box plot for the different data preprocessing methodologies.

As shown in the fig 1 and 2 Data

B. *Exploration Testing Results General Hyperparameters*

test_size	batch_s	epochs	patience	min_d	starting_h_l_size
0.2	8	165	10	0.01	64

num_h_l	h_l_act	out_act
4	tanh	softmax

View Notebook for more data plots. [2]

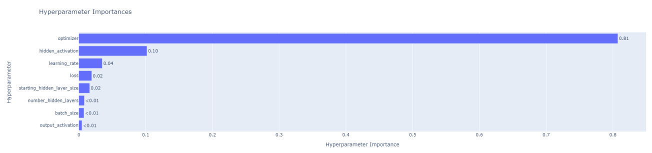

Fig. 3. Test Parameter Importance.

The optimizer and hidden activation function have the highest weighting. This is because the optimizer used completely determines how well the model is trained and as such how well it would perform. Additionally, the hidden layer activation function has a high importance, because this will determine how well the model generalizes with a specific training algorithm. The number of hidden layers and the size of the hidden layers have little effect on the model. While there is an increase in performance, increasing these values is negligible, and therefore, to make the model smaller, a hidden layer starting size of 64 and a number of hidden layers of 4 will be used for all subsequent models. Batch size has some effect on the model because it determines the loss value. Larger batch sizes can lead to larger loss values, which can mean that during training, the model oscillates more.

C. *Exploration Testing Results RProp*

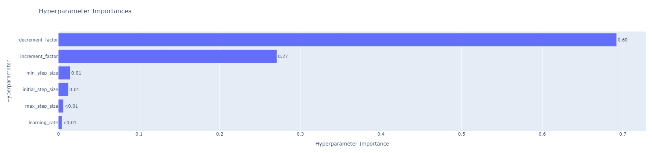

Fig. 4. Test Parameter Importance.

initial_step_size	increment_factor	decrement_factor
0.001078672	1.1141238793860642	0.3694139226122018

min_step_size	max_step_size
0.000599729	68.24850538411529

For RProp, increment and decrement factors are of the highest importance. This makes sense because these two determine the oscillation of the training algorithm.

D. *Exploration Testing Results Optimizers*

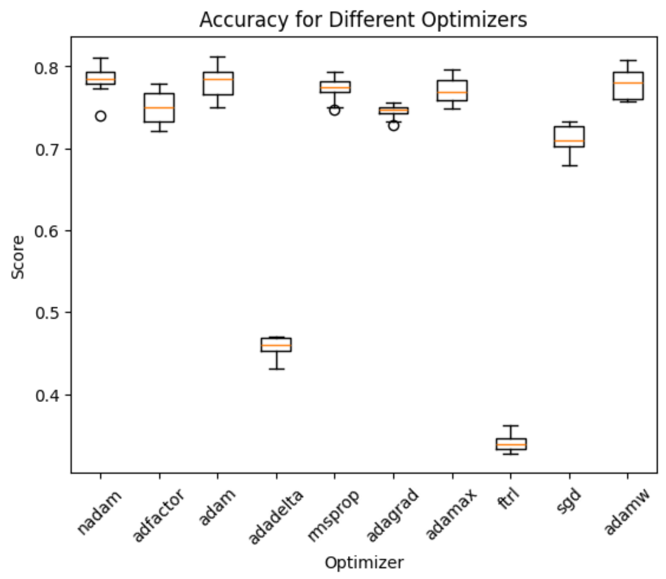

Fig. 5. Box Plot for Diff. Optimizer performance.

Nadam, Adam and Adamw seem to perform the best after tuning. Adadelta and ftrl perform the worst. The other optimizers are close in berformance to the best 3. Nadam has the best overall performance, but only by a small margin.

E. Final Testing Results Nadam

Some overfitting is present in the model, but it is not significant. The model is able to generalise well to the test data. The learning rate has a significant effect on the model's performance. Beta 1, Beta 2, loss function and hidden layer activation function have a moderate effect on the model's performance. Epsilon has very little effect on the model's performance. [2]

F. Final Testing Results All

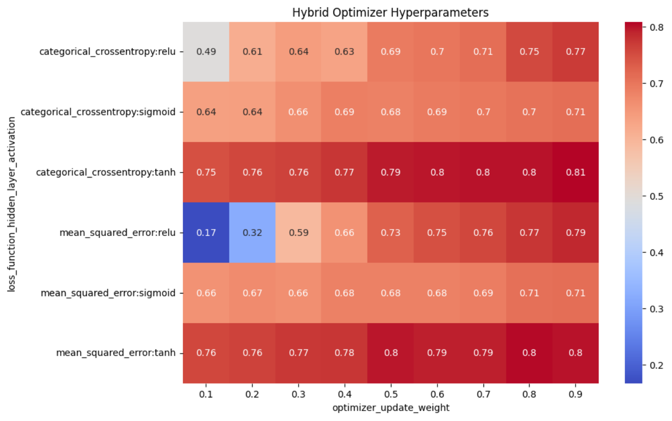

Fig. 6. Heatmap for the Hybrid Optimisation algo. over 200 epochs.

Heat Maps for the 3 optimizers over 50 epoches

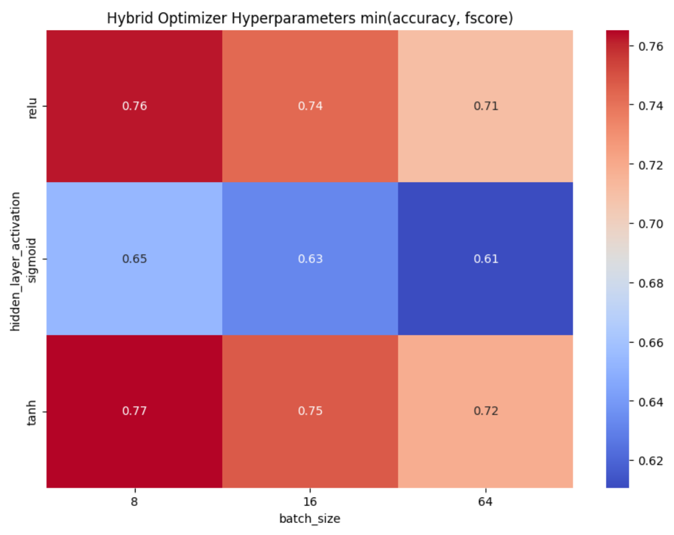

Fig. 7. Hybrid Heatmap

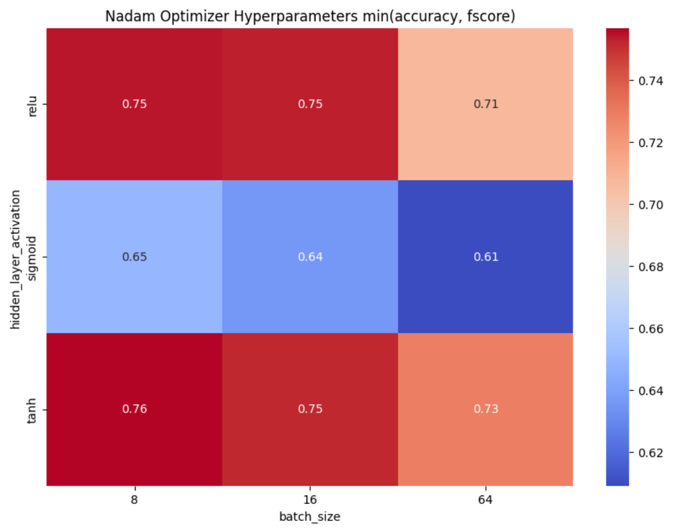

Fig. 8. Nadam Heatmap

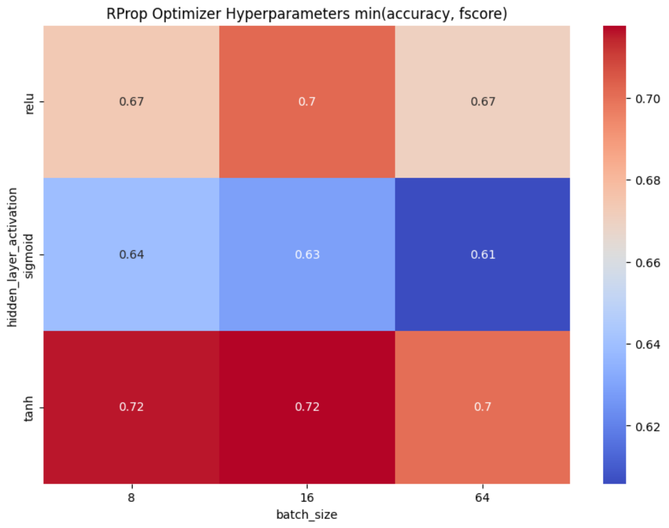

Fig. 9. RProp Heatmap

Final Results

Optimizer	accuracy	fscore	accuracy_std	fscore_std	epochs
Hybrid	0.8082437275985663	0.8088539923937008	0.016832739	0.016975323	165
RProp	0.7912186379928315	0.7919421719709687	0.003075158	0.002853409	200
Nadam	0.839605735	0.8395578090942438	0.005745336	0.005608286	200

V. Conclusion

As shown by the heatmap for hybrid optimization, the Nadam optimizer still performs the best and TanH activation function also generally performs the best. This is likely due to the fact that Nadam is better at managing oscillation than RProp.

Overall, the hybrid optimization does not perform as well as Nadam on its own, this is likely due to the fact that the 2 optimizers weight update values can point in opposite directions and with RProp seemingly doing worse on average, this would hinder Nadam. However, the hybrid optimization does seem to produce more stable results as the standard deviation of the accuracy and Fscore is lowest here.

References

[1] Gómez, I., Franco, L. & Jerez, J.M. Neural Network Architecture Selection: Can Function Complexity Help?. *Neural Process Lett* **30**, 71–87 (2009). https://doi-org.uplib.idm.oclc.org/10.1007/s11063-009-9108-2

[2] Jordaan, D. *Almond-Classification-COS-711-Assignment-2-2024* https://github.com/DamianJordaan/Almond-Classification-COS-711-Assignment-2-2024