COS711 Assignment 2

Dino Gironi
u21630276
dino.gironi@up.ac.za

Github Repo: https://github.com/Rec1dite/AImond

***Abstract*—This report explores the performance of different neural network hyperparameter configurations for the problem of almond classification. The dataset consists of 12 attributes derived from images of 3 types of almond: *Mamra*, *Sanora*, and *Regular*, totaling nearly 3000 entries. [9]**

A standard methodology is followed to prepare and train the network, including data preprocessing, and, splitting data for testing/training, defining architecture & training the network, k-fold cross-validation and evaluation across multiple hybrid-optimization models. Grid search is applied across multiple configurations to identify the most effective architecture and configuration for the network.

Finally, a hybrid-optimization network constructed from the insights of the search algorithms and statistical tests is constructed to solve the problem.

The conclusion found by this experimentation is that hybrid optimization, with multiple neural network optimizers working in unison to compute each weight update, can improve overall performance of the network, even when tested on unseen data. Other conclusions about additional hyperparameters were reached, including which specific combinations work well with each other to produce the best overall performance.

I. Introduction

This assignment focuses on the classification of different types of almonds using neural networks. The goal is to design and train a machine learning model that can accurately identify almonds as belonging to one of three categories: *Mamra*, *Sanora*, or *Regular*.

The dataset used contains 12 numeric attributes describing physical features derived from top-down images captured of 2803 different almonds. The most prominent of these include primary characteristics such as length, width, and thickness features, as well as area and perimeter. Secondary characteristics are formulaicly derived, such as roundness, solidity, and eccentricity.

As discussed in subsection III-A, the dataset is thoroughly pre-processed and cleaned-up for consistency and removal of missing values, as well as making it more interpretable for the network (having more predictable structure).

After the data preparation phase, the network will be defined in a structured, modular manner, enabling easy reconfiguration and testing of different architectural structures and hyperparameter values.

The performance of neural networks can vary widely based how they're configured, so hyperparameter search techniques like grid search are employed to find the best combination for optimal performance on the testing benchmark set. This process will enable empirical identification of which settings lead to more effective classification.

The final part of the report involves a comparative analysis of the various training algorithms explored, including a hybrid method that combines the strengths of multiple approaches to enhance classification accuracy.

II. Background

A. Multi-layer Perceptron

To address the classification task at hand, a **multi-layer perceptron** is employed - a type of feed-forward neural network consisting of an input layer, n hidden layers, and an output layer. In my specific implementation, I have incorporated a smaller 'tapered' hidden layer that resemble an encoder/decoder network structure. This is a well-known design choice which aims to force the network to extract more meaningful and compact representations of the input data, and which through repeated testing I have found to be measurably effective at increasing classification accuracy, compared to a 'regular' perceptron where all layers are of equal dimension. [2]

In this assignment I attempt to optimize for various hyperparameters, most prominently **activation functions**, **optimizers**, and **loss functions**, discussed below.

B. Activation functions

Activation functions introduce non-linearities into the layered structure, enabling the modelling of non-linear relationships in the data. It is through the layer of multiple activation functions accepting linear inputs, that a neural network is enabled as a 'universal function approximator'. Among the most commonly used modern activation functions is the Rectified Linear Unit (1) function, which outputs the input directly if it is positive; otherwise, it outputs zero.

$$ReLU(x) = max(0, x) \tag{1}$$

This function offers several advantages, including ease of computation and reduced likelihood of the vanishing gradient problem, making it popular in modern neural network architectures. [4] Other activation functions such as Sigmoid and Tanh are used less frequently due to their potential downsides, including saturation issues that can hinder learning, but they are sometimes employed in specific contexts where their properties are beneficial. They are still tested in the below experiments nonetheless as a base of reference.

C. Optimizers

Optimizers are algorithms that adjust the weights of a neural network during training to minimize the loss function. The *learning rate* of an optimizer is a key parameter which determines the step size by with which the weights are updated each iteration.

Adaptive learning rate methods, like the *Resilient Backpropagation* (`RProp`) algorithm, are particularly and relevant to the experiments conducted below. RProp enhances the standard backpropagation approach by considering the sign of the gradient rather than its magnitude, allowing it to adaptively adjust the learning rate for each weight based on the consistency of the gradient's direction. This generally results in faster convergence and improvement in performance, particularly useful in scenarios with oscillatory behavior or varying scale of gradients. [1]

The steps involved in computing `RProp` weight updates are shown here for illustration:

1) Weight update:

$$w_{i,j}^{(t+1)} = w_{i,j}^{(t)} - \Delta w_{i,j}^{(t)}$$

2) Adaptation of the update value:

$$\Delta_{i,j}^{(t+1)} = \begin{cases} \eta^+ \cdot \Delta_{i,j}^{(t)}, & \text{if } \frac{\partial E}{\partial w_{i,j}^{(t)}} \cdot \frac{\partial E}{\partial w_{i,j}^{(t-1)}} > 0 \\ \eta^- \cdot \Delta_{i,j}^{(t)}, & \text{if } \frac{\partial E}{\partial w_{i,j}^{(t)}} \cdot \frac{\partial E}{\partial w_{i,j}^{(t-1)}} < 0 \\ \Delta_{i,j}^{(t)}, & \text{otherwise} \end{cases}$$

3) Gradient step determination:

$$\Delta w_{i,j}^{(t)} = \begin{cases} -\Delta_{i,j}^{(t)}, & \text{if } \frac{\partial E}{\partial w_{i,j}^{(t)}} > 0 \\ +\Delta_{i,j}^{(t)}, & \text{if } \frac{\partial E}{\partial w_{i,j}^{(t)}} < 0 \\ 0, & \text{otherwise} \end{cases}$$

Another fashionable optimizer is Adam (Adaptive Moment Estimation), which combines the advantages of both AdaGrad and RMSProp. Adam is commonly used due to its efficient handling of sparse gradients and its ability to adapt its learning rate over the course of training, making it generalizable and suitable for a wide range of problems. [3]

D. Loss functions

Loss functions are vital in the training process - determining the delta between target values and predictions. The choice of the loss function can significantly impact the learning process and the model's final performance, as it is the fundamental function which guides the algorithm's search process. A common loss function for classification tasks is *categorical cross-entropy*, which measures dissimilarity between a predicted probability distribution and the truth, particularly prominent in multi-class settings, such as the almond classification problem at hand. [5]

E. Maintaining the Integrity of the Specifications

III. Experimental Setup

The Jupyter notebook is available in the following Github repo: https://github.com/Rec1dite/AImond

The neural network was created using the Pytorch framework for Python, using a standard `Sequential` model with 4 hidden layers, designed in such a way that many hyperparameters such as choice of *activation* and *loss* functions are externally configurable each run. The full pipeline is described in the sections below:

A. Data Preprocessing

The preprocessing stage involves several important tasks aimed at preparing the almond dataset for analysis and consumption by the network/s. My focus hereby was to maintain the integrity of the data, while addressing

1) Column names are standardized for easier reference and the redundant/irrelevant `Index` column is removed.
2) The categorical `Type` target values are converted to a one-hot encoding format over 3 columns to remove any form of enumerability or 'order' relation between categories which may be misconstrued when using a single numerical attribute.
3) The `Length`, `Width`, and `Thickness` columns are isolated. Missing values are replaced with a value of -1, and then the values within each row across these three columns are sorted. This sorting process produces 3 new columns, where the first contains only -1's (subsequently discarded), and the second two contain dimensional data which is reinterpreted as the new `Length` and `Width`, discarding `Thickness` altogether.
4) To prevent the loss of potentially meaningful data, the original 'missing' dimension within each row of the dimensional data is recorded in a one-hot encoded set of columns. This effectively represents *the column which was removed in by the processing in the above step*. This new format of columns still contains all the same data as the original 3 dimensions, however it is now formatted in a much more consistent manner, with missing values removed (without needing to use imputation).
5) Derived geometric properties from the dataset, which also had missing values, are recomputed using the new `Length` and `Width` columns, via their original formulae as provided by the dataset curators. These properties include `Roundness`, `Aspect Ratio`, and `Eccentricity`.
6) Finally, the features and targets are separated into `X` and `Y` dataframes for modeling purposes during training.

In all, the above preprocessing system produces consistent data with no missing values, and without requiring the use of any statistical data imputation techniques.

B. Splitting Test/Training

As seen in the Jupyter notebook, multiple different types of runs were performed, with different mechanisms for the

splitting of data. In the simple, single network runs (for the sake of only measuring convergence and performance), no validation set was used, and just a simple *80:20* split was performed into training and testing sets respectively.

For more thorough runs with aggregated results, such as the grid search algorithms, *K-fold cross validation* was performed, primarily with a k value of 5 and 10, depending on the number of other search parameters that were also considered for each set of runs.

C. Constructor functions

To achieve a high degree of composability and modularity, a series of constructor functions are defined in the Jupyter notebook which parameterize each step of the network construction, including specifying architectural values, running backpropagation epoch loops, testing & evaluating each model, performing cross-validation steps, and plotting the results.

By structuring the code in this way, we enable simple and reproducible experimentation, as well as a smoother fine-tuning process, since the network itself is completely abstracted away, and only the hyperparameters and metrics remain to be defined by the caller.

D. Network architecture

A simple multi-layer feedforward neural network was used, with a small twist: Taking inspiration from *encoder-decoder* neural networks, whose architectures are tapered inwards to force the neural network to learn more distinct features from the dataset, I parameterized the network such that the middle hidden layer was of smaller dimension than the surrounding hidden layers. Through some experimentation, I did find that this seemed to perform at lesat on-par, but occasionally better than a straightforward network where all layers were equal size. Thus this architecture was kept for the final network design. I have not spent time attempting to optimize this architecture through a formal search process, as there were enough other hyperparameters to search already, however I do expect that some more performance can potentially be gained by playing more with such an approach, and fine-tuning the overall architecture on the level of individual layers.

E. Hyperparameter considerations

This report primarily studies the effect of 4 network hyperparameters on the performance of the network for almond classification. These include *activation functions*, *loss functions*, *optimizers*, and *learning rates*. The set of each of these was chosen in part from experimentation and iterative testing, however in the case of loss functions, the choice was primarily given by the functions having good compatibility with the *one-hot* encoding of the output layer, since many loss functions are not designed for such an encoding, and thus give poor or inaccurate metrics for the current error.

Activation functions were selected from a set of commonly used functions; Popular choices include `ReLU`, `Leaky ReLU`, and `Sigmoid`, each having their own strengths and weaknesses. For instance, `ReLU` is widely used and known to be effective despite its simplicity, however certain issues such as the "dying ReLU" problem can hinder performance in certain cases, which is mitigated by alternatives such as `Leaky ReLU`. [7]

Optimizers were assessed based on their convergence rates, with earlier-converging optimizers being favoured overall. More sophisticated optimizers included with Pytorch were thrown in, in particular `Adam` and its variants, since the adaptive learning rate algorithms enable more robust handling of noisy data and better overall convergence. Conversely, simpler optimizers like `SGD` were considered for their stability in certain scenarios, despite requiring extensive tuning.

Learning rates in the grid search were selected simply in an approximate range of value near zero, with each having one order of magnitude's difference between them.

Overall, each hyperparameter was scrutinized across a series of informal experiments, before being chosen to be added to the set for optimization.

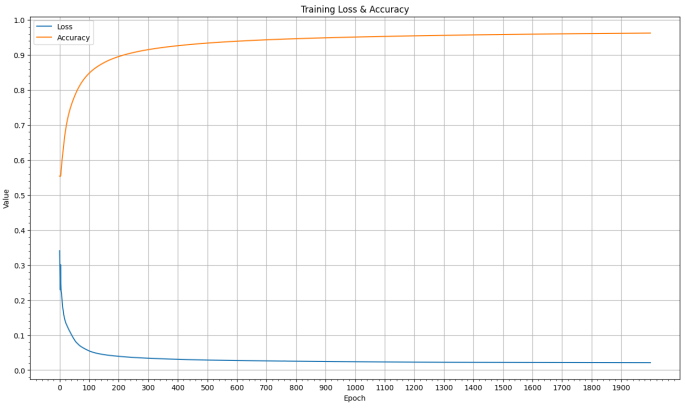

Fig. 1. Simple run of the network using RProp optimizer with MSE loss

IV. Research Results

A. Optimizer & Loss Function Grid Search

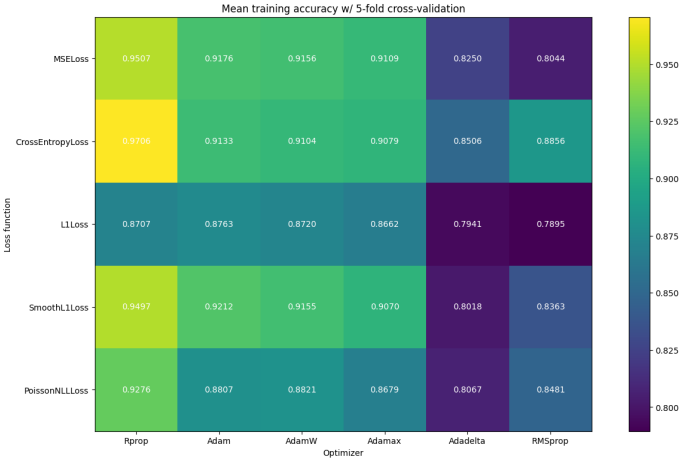

Fig. 2. Grid search for mean final training accuracy for different combinations of optimizers and loss functions

Fig. 3. Grid search for mean testing accuracy for different combinations of optimizers and loss functions

Figures 9 and 10 each depict a grid search across a number of optimizers and loss functions.

1) Optimizers: In terms of optimizers in figure 9, the `RProp` seems to show the most promissing results, obtaining the highest overall accuracies across all optimizers, however in figure 10 we see that the performance drops off significantly, only providing mediocre results in comparison to `Adam`, `AdamW`, and `Adamax`. This seems to imply that the `RProp` optimizer may be prone to overfitting under this specific configuration, potentially due to the fact that, unlike `Adam` and variants, `RProp` does not contain any built-in regularization mechanisms, like *L2 regularization* or *weight decay*, which help improve generality by dynamically restraining optimization throughout the run. Additionally Rprop does not have any sense of 'momentum', like Adam optimizer variants, which helps to dampen oscillations and promote steadier convergence and better generalization. The inclusion of momentum helps optimizers navigate the loss landscape more smoothly and avoid getting stuck in 'steep' local minima that can lead to overfitting or premature convergence. [3]

The worst performing optimizer across testing and training is `Adadelta`, suggesting that it may struggle to find optimal parameter updates consistently across different datasets or model configurations. This is likely due to its aggressive learning rate adaptation, which can cause inefficiencies or slow convergence during optimization. [6]

An interesting outlier is the combination of `MSELoss` with `RMSProp`, which performs poorly (yet consistently) on both training and testing sets at *80.85%* for the latter. This may be because this specific pairing leads to sub-optimal gradient updates, and `RMSProp` is not ideally suited for regression tasks where `MSELoss` is typically applied, possibly resulting in noisy gradient approximations.

Overall, the best optimizer across all loss functions appears to be `AdamW`, with an exceptional average test score of *88.96%* when paired with `MSELoss`. This combination likely benefits from `AdamW`'s inherent weight decay which contributes to better generalization and stable convergence, especially when dealing with continuous output spaces, such as that generated by `MSELoss`.

2) Loss Functions: In terms of loss functions, `L1Loss` clearly performs worst overall, on both training and testing. A likely reason is that `L1Loss` does not work well with one-hot encoding of the output layer, due to its loss landscape which can introduce non-smooth gradients for models that output discrete or binary categories, leading to unstable learning dynamics. [8] Additionally, `L1Loss` tends to be less forgiving with outliers compared to `SmoothL1Loss` or `MSELoss`, which can result in higher error sensitivity during backpropagation.

The best overall loss function via average accuracies is `SmoothL1Loss`, AKA *Huber loss*. It amalgamates advantages of both `L1Loss` and `MSELoss`, allowing it to be less sensitive to outliers compared to `MSELoss`, while maintaining a degree of robustness in gradient computation similar to `L1Loss`. This makes `SmoothL1Loss` particularly useful in scenarios where the dataset contains a lot of noise, which can veer the training process in the wrong direction. Its robustness makes it a good choice for tasks requiring both precision and stability.

Overall, performance across all models is relatively consistent, with variance of approximately *1%* in accuracy on the training and testing set. This is likely because the number of epochs used (*1000*) is sufficient such that all the models approximately settle into a similar configuration with time.

Additionally, the test results suggest that the loss functions' inherent characteristics and their interaction with different optimization strategies significantly influence the convergence behavior and final model accuracy, which underscores the importance of selecting compatible pairs of optimizers and loss functions during network design.

B. Comparing Optimizers & Hybrids

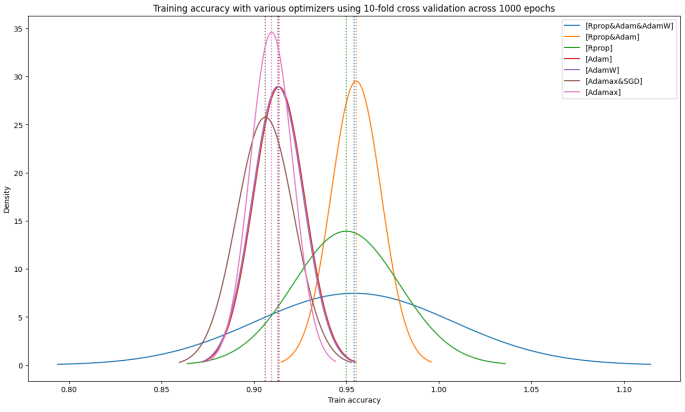

Fig. 4. Training accuracy bell curves across optimizers

Figures 4 and 5 depict the comparative training accuracy and standard deviation of various optimizers, each running with a basic neural network with `MSELoss` loss function and `ReLU` activation function.

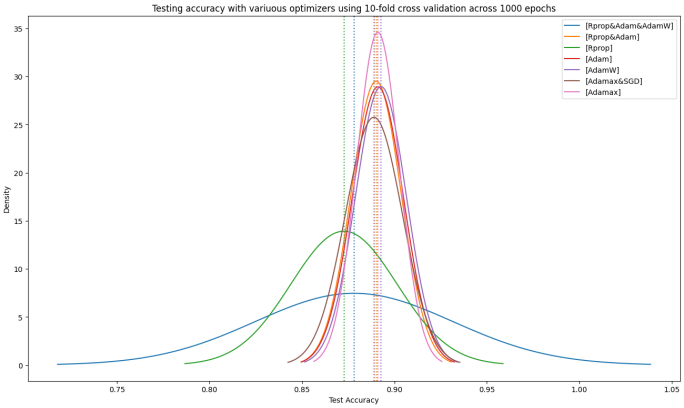

Fig. 5. Testing accuracy bell curves across optimizers

Training results in figure 4 appear tightly correlated between two groups, each hovering around *91%* and *95%* accuracy respectively. Notably, the higher training accuracy group of models consists of two of the hybridized models. This seems to imply that hybridization can result in better fitting of the models to the training set, however - as seen by the high variance score for the `[RProp&Adam&AdamW]` hybridized optimizer - the downside of this is that it may lead to less consistent and reliable performance from the model each time it is re-trained.

Interestingly, on the testing set, the difference in training performance seems to become much less relevant, with all models performing nearly identically around 88% accuracy.

Notable outliers are the `RProp` and `[RProp&Adam&AdamW]`, with much higher variance scores, and slightly lower testing accuracy than the other optimizer models. This would imply that the hybridization of these optimizers seems to result in less consistent performance across runs for the models, desipite the average training accuracy remaining good.

In contrast, the `RProp&Adam` optimizer seems to perform very well.

Conflicting optimizers averaging their weight updates likely causes the network to converge a lot slower, without necessarily maintaining the ability to escape local optima. This appears to result in poorer generalizability, and also more varied results across runs, since the algorithm fails to adequately find local optima.

In summary, it does appear promising that just the right combination of optimizers working together can produce an optimizer which performs better than the sum of its parts, however the wrong combination can have the exact adverse effect, and degrade reliability and performance.

Section IV-D explores the set of all possible combinations of the best optimizers found by the grid search algorithm, and attempts to find a combination which performs better than any of the individual optimizers in isolation.

Fig. 6. Grid search for mean final training accuracy for different combinations of learning rates and activation functions

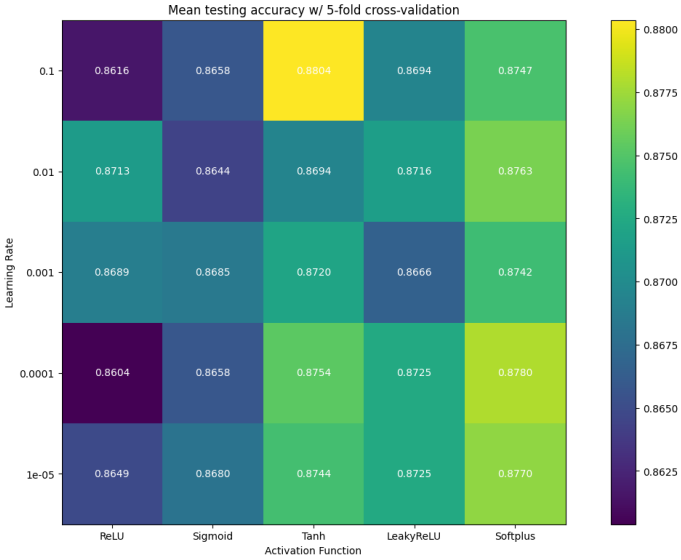

Fig. 7. Grid search for mean testing accuracy for different combinations of learning rates and activation functions

C. Learning Rate & Activation Function Grid Search

Figures 6 and 7 measure the accuracies across a grid search of learning rate steps for the `RProp` optimization function, in combination with a variety of different activation functions.

Performance in these heatmaps is remarkably consistent for different learning rates across all runs - all mean accuracy scores falling within *2%* of each other across both grids - potentially alluding to the resilience of the `RProp` optimizer for effectively solving the problem under different constraints. The fact that 1000 epochs were used for each of these runs may account for this, in that learning rates that caused slower convergence still perform on-par with faster learning rates due

Fig. 8. Standard deviation of testing accuracy for different combinations of learning rates and activation functions, corresponding to figure 7

to there being sufficient epochs for all rates to eventually converge on an effective optimum.

This is further confirmed by figure 8, plotting the testing accuracy *standard deviations* for each corresponding grid point in figure 7. We see that these deviations (and thus, the variance in accuracy scores for each point) are very low across the entire plot, implying very consistent results across all *k-fold* runs.

D. Hybrid Optimization Algorithm

TABLE I
FINAL TESTING ACCURACY OF DIFFERENT OPTIMIZER COMBINATIONS OVER 500 EPOCHS WITH 5-FOLD VALIDATION

Optimizer Combination	***Accuracy* ± *Std.***
Rprop	0.8673 ± 0.0125
Adam	0.8844 ± 0.0155
AdamW	0.8882 ± 0.0105
Adamax	0.8851 ± 0.0115
Rprop&Adam	0.8818 ± 0.0109
Rprop&AdamW	0.8875 ± 0.0153
Rprop&Adamax	0.8785 ± 0.0149
Adam&AdamW	0.8820 ± 0.0109
Adam&Adamax	0.8835 ± 0.0127
AdamW&Adamax	0.8839 ± 0.0130
Rprop&Adam&AdamW	0.8901 ± 0.0110
Rprop&Adam&Adamax **	0.8913 ± 0.0117
Rprop&AdamW&Adamax	0.8861 ± 0.0144
Adam&AdamW&Adamax *	0.8975 ± 0.0146
Rprop&Adam&AdamW&Adamax	0.8806 ± 0.0092

* Best accuracy
** 2nd-best accuracy

To derive the final hybrid optimization strategy for the network, all the best hyperparameters as discovered in the above grid searches were applied to a model, aside from the set of optimizers.

For the optimizers, the top 4 best performers discovered by the grid search - namely `RProp`, `Adam`, `AdamW`, and `Adamax` - were batched in all possible combinations and tested with 5-fold validation. The results are shown in table I, with the overall best performing optimizer - with a mean score of **89.75%** testing accuracy

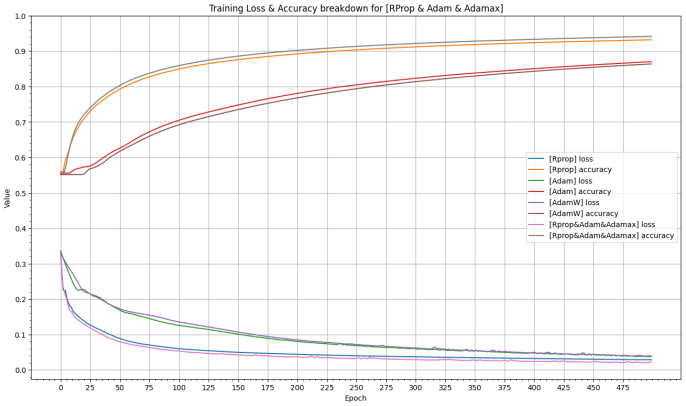

Fig. 9. Breakdown of individual contributions from a single run of optimizers for [RProp & Adam & Adamax]

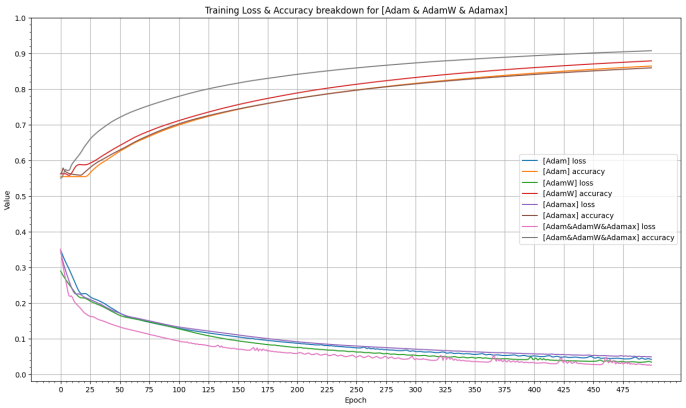

Fig. 10. Breakdown of individual contributions from a single run of optimizers for [Adam & AdamW & Adamax]

The conclusion reached by this experiment is that it is indeed possible for the hybridisation of optimizers to perform better than the individual optimizers in isolation. However, it seems that this combination of optimizers needs to be carefully selected, as the opposite effect can also happen, as seen in the case of the `[RProp & Adam & AdamW & Adamax]` optimizer, which performs worse than nearly all of its constituents.

Another point worth noting is that the results obtained here may not be entirely statistically significant, given that there is a fair bit of overlap between the variance in the accuracy distributions of each optimizer combination. While the results are averaged over numerous runs, and some have performed better on average, there is a small statistical chance that that certain optimizers may have started with more favorable conditions, than others, and thus had their mean accuracies slightly boosted above the true value.

1) Performance T-tests: To demonstrate this, a statistical significance test of the difference in between the best performing optimizer combination `[Adam & AdamW & Adamax]` and the worst individual optimizer `RProp` was performed, as seen in the last cell in the Jupyter notebook.

The mean accuracy for `[Adam & AdamW & Adamax]` was recorded at **89.75%** with a standard deviation of **0.0146**. `RProp` achieved a mean accuracy of **86.73%** with a standard deviation of **0.0125**.

The t-test yielded a p-value of **0.002**, which is below the commonly accepted threshold of 0.05 for statistical significance. The t-test yielded the following results:

- T-statistic: 3.51
- P-value: 0.0079
- Degrees of Freedom: 8

P-value: The p-value is very small (< 0.01). This indicates that the difference in accuracy between the two optimizer configurations is statistically significant. This result strongly suggests that the observed performance improvement from the hybrid optimizer is statistically significant, and thus enhanced accuracy is not just due to favorable conditions or random chance.

A t-test between the best and second best (`[RProp & Adam & Adamax]`) performers however, indicates a likely chance that the difference in results is not statistically significant for the number of runs performed.

- T-statistic: 0.74
- P-value: 0.4798
- Degrees of Freedom: 8

2) Comparing hybrid optimizers with singular standalone optimizers: The comparison of hybrid optimizers with individuals may reveal interesting insights into the dynamics of neural net optimization. The improved performance of hybrid optimizers suggests that the blending of adaptive and momentum-driven strategies inherent in these optimizers can complement each other, potentially reducing the drawbacks associated with using a single technique - similar to the effect witnessed in ensemble machine learning techniques, where multiple whole models are used to arrive at a singular prediction. Hybrid setups can harness the 'consensus' of multiple optimizers to provide a more rounded optimization paradigm, with the potential to mitigate overfitting and improve robustness against noise and local optima.

Another point which warrants further investigation, is the effect of hybrid combinations on the interplay of internal optimizer mechanisms: factors such as learning rate adaptation, weight regularization, and gradient accumulation that each optimizer moderates must interplay somehow, and their potential conflict is what may also produce worsening of hybridized results in some of the above test cases.

V. Conclusions

The results found above have provided a comprehensive analysis of the impact various hyperparameter choices can have on the performance of a neural network for the problem of almond type classification. It is clear that the importance of choosing effective combinations of the various hyperparameters is paramount to training an effective network for the task at hand.

A. Findings

The main finding from these experiments is the significant effect of optimizers on model performance. As expected, the traditional modern optimizers like `Adam` and `AdamW` achieved high accuracies, while simple heuristic optimizers such as `RProp` performed less well on testing set, but competitively on the training set (suggesting they may be more prone to overfitting). The more interesting conclusion reached, however, lies in the performance of hybrid optimizer combinations: The hybrid approach, specifically the combination of `Adam`, `AdamW`, and `Adamax`, resulted in the highest recorded testing accuracy across all runs of the algorithm that I've done, illustrating the potential advantages of leveraging complementary strengths of multiple optimizers. However, this technique requires careful selection to avoid combinations that degrade performance, as seen with the `[RProp & Adam & AdamW & Adamax]` ensemble.

B. What was learned

The investigation into loss functions highlighted the potential versatility and effectiveness of `SmoothL1Loss` and `MSELoss`, particularly when paired with advanced optimizers like `AdamW`. These findings further demonstrate importance of aligning loss functions with the network's specific learning requirements, particularly when data may be noisy or contain many outliers.

The implications of statistical significance were also considered throughout different sections in the report. The variances observed among accuracy measurements highlight that meticulous experimental design is often necessary, particularly in situations where many different hyperparameter variations can produce very similar results. Repeated trials are thus essential to ensure robust conclusions. Although the results point toward certain trends and conclusions, factors such as random initialization and dataset peculiarities could introduce variations that slightly skew the outcomes.

C. Future work

Future work has many potential directions to explore which may enhance the understanding and application of the optimization strategies investigated above. One particularly notable avenue is how adaptive learning rate mechanisms, such as those seen in `Adam` and variants, can be integrated to work in tandem with hybrid optimizer approaches, as at the moment it is likely that these adaptive techniques may conflict with each other on a low level when weight updates are averaged between several optimizers simulatenously.

Another point which future work may improve on is that of visualization. I found in some instances of graphing model performance, it was difficult to depict multiple aspects of the performance simulatenously, such as simulatenous mean and

standard deviation on a grid plot, and I was forced to reduce the dimensionality or split the graph into two separate plots. The development of more sophisticated graphical plotting tools could drastically help with the interpretation of results and with determining statistical significance of performance improvements across models.

D. Closing

In summary, this research reinforces the importance of a well-thought-out approach in hyperparameter selection and optimization strategy development, showcasing the enhanced potential realizable through a combination of theoretical understanding and empirical evaluation. The promising results obtained from hybrid optimization strategies open up new avenues for future research on adaptive and flexible optimization frameworks in machine learning.

VI. Bibliography

References

[1] Ruder, S., 2016. An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

[2] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25.

[3] Kingma, D.P., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[4] Glorot, X., Bordes, A. and Bengio, Y., 2011, June. Deep sparse rectifier neural networks. In Proceedings of the fourteenth international conference on artificial intelligence and statistics (pp. 315-323). JMLR Workshop and Conference Proceedings.

[5] Bishop, C.M. and Nasrabadi, N.M., 2006. Pattern recognition and machine learning (Vol. 4, No. 4, p. 738). New York: springer.

[6] Zeiler, M.D., 2012. ADADELTA: an adaptive learning rate method. arXiv preprint arXiv:1212.5701.

[7] Xu, J., Li, Z., Du, B., Zhang, M. and Liu, J., 2020, July. Reluplex made more practical: Leaky ReLU. In 2020 IEEE Symposium on Computers and communications (ISCC) (pp. 1-7). IEEE.

[8] Qian, W., Yang, X., Peng, S., Yan, J. and Guo, Y., 2021, May. Learning modulated loss for rotated object detection. In Proceedings of the AAAI conference on artificial intelligence (Vol. 35, No. 3, pp. 2458-2466).

[9] Moradi, S. (n.d.) Almond Types Classification. Available at: https://www.kaggle.com/datasets/sohaibmoradi/almond-types-classification (Accessed: 4 October 2024).