Exploration of Optimisation Methods Based on Fractional Derivatives in Neural Networks

Diogo Marto NºMec 108298
DETI, Universidade de Aveiro
Aveiro, Portugal
diogo.marto@ua.pt

Inês Matos NºMec 124349
DETI, Universidade de Aveiro
Aveiro, Portugal
inessmatos@ua.pt

***Abstract*—This project explores a novel optimization approach based on fractional gradients—an extension of traditional gradient descent that leverages non-integer (fractional) derivatives with a dynamically adjusted order. The primary objective is to evaluate whether this method can enhance convergence speed and performance in training neural networks. Building upon the ψ-Hilfer derivative framework, we implement several fractional-order gradient descent algorithms and integrate them into neural network training pipelines. The proposed methods are benchmarked against standard optimizers using datasets such as MNIST, CIFAR-10, and HappyFace. Experimental results demonstrate that while fractional optimizers offer no significant advantages in simple tasks, they outperform traditional methods in challenging or ill-conditioned scenarios, suggesting their potential as a robust alternative in complex optimization landscapes.**

***Index Terms*—Fractional Derivatives, Numerical Methods, Optimization Algorithms, Neural Networks**

This report was made for the CAA course with Petia Georgieva (petia@ua.pt) as its instructor.

I. Introduction

Optimisation is a fundamental tool in many computational problems, including applications in machine learning and mathematical simulation. It's not an understatement to say that optimisation is one of the keystones that powers the AI revolution. Gradient descent is often used to find a function's minima. Still, on the more "complex" problems, for example, if the function has many local minima or is irregular, these methods may struggle with convergence or fall into suboptimal solutions, especially in high-dimensional and irregular landscapes common in machine learning.

This study area remains an open problem, and many approaches are being explored. This report examines one such approach in the context of neural networks: using fractional gradient methods via ψ-Hilfer derivative proposed in [1]. This technique is based on non-integer derivatives, an extension of the classical concept of derivatives coupled with a trained approach of the fractional-order of the gradient.

Its integration into neural networks remains untested. Thus, this report aims to document the implementation and present the results of a class of algorithms based on the original paper [1]. The effectiveness of these algorithms will be tested on real-world classification tasks using standard datasets such as MNIST and CIFAR-10.

This work aims to contribute to understanding fractional methods' potential in optimisation contexts by exploring their benefits and limitations. The report is organized as follows: the next section presents the current state of the art in this field, followed by the theoretical framework of the problem, and a description of the implemented algorithms. This is followed by the experimental results and their discussion, concluding with conclusions and future work.

II. State of the Art

Gradient optimisation has long been a fundamental tool in computer science, particularly valued for its role in machine learning and numerical computation. Thanks to their simplicity and efficiency, Gradient Descent variants such as SGD (Stochastic Gradient Descent), Adam, and RMSprop have become standard tools in training machine learning models [2].

However, classical gradient methods also have some limitations, particularly in terms of problems that arise in complex error surfaces. Therefore, it is essential to look for alternative approaches, such as fractional derivatives. These seek to explore richer properties of the objective function.

Fractional derivatives are a mathematical extension of traditional derivatives, offering greater flexibility in modelling phenomena with non-local, memory-inherent nature behaviour. Thus, thanks to the fact that they allow non-integer orders of differentiation, this approach has gained visibility in areas such as signal processing, dynamical systems, and, more recently, in optimisation algorithms [3], [4]. Furthermore, several theoretical studies have also indicated that using fractional gradients can improve the performance of optimization methods in terms of stability and the ability to escape local minima [5].

Some works implementing fractional gradients in neural networks already exist, namely:

[6] Sheng et al. proposed a fractional-order gradient method specifically for the backward propagation of convolutional neural networks. To address the issue of fractional-order gradient methods not converging to a real extreme point, they designed a simplified fractional-order gradient method based on Caputo's definition. In their method, parameters within layers are updated using the designed fractional gradient method, while propagations between layers still utilize integer-order gradients. Practical experiments demonstrated fast convergence, high accuracy, and the ability to escape local optimal points.

[7] Lou et al. proposed a variable-order fractional gradient descent optimization algorithm that generalizes the classical

gradient descent method by introducing a variable-order fractional derivative. In their approach, the derivative order is adjusted with the number of iterations, and the convergence of this algorithm was thoroughly analyzed. The proposed method was applied to the training of fully connected neural networks. The results indicated that the proposed method achieved a faster learning rate and higher accuracy on training and validation sets.

[8] Shin et al. introduced a class of fractional-order optimization algorithms, defining a fractional-order gradient using Caputo fractional derivatives. Replacing integer-order gradients with these fractional-order gradients, they derived concrete algorithms such as the Caputo fractional GD (CfGD) and the Caputo fractional Adam (CfAdam). The superiority of CfGD and CfAdam was demonstrated on several large-scale optimization problems in scientific machine learning, including ill-conditioned least squares problems with real-world data and the training of neural networks with non-convex objective functions. Numerical examples showed that CfGD and CfAdam resulted in acceleration compared to their integer-order counterparts (GD and Adam, respectively).

[9] Harjule et al. proposed a modified fractional gradient descent algorithm by integrating the benefits of a metaheuristic optimizer. Their detailed convergence analysis indicated that the method enables a more gradual and controlled network adaptation to the data. A significant contribution of their work is the empirical determination of the optimal fractional order for each dataset, which significantly impacts neural network training with fractional gradient backpropagation. Empirical results demonstrated that this optimizer with optimal order yields more accurate results than existing optimizers.

[10] Chen and Xu introduced λ-FAdaMax, a fractional-order gradient descent method that builds upon the Caputo fractional derivative and the Adam framework. They found that the second moment is a primary factor influencing both precision and speed, where using a fractional-order gradient for the second moment improves precision but decreases speed, and vice versa for a first-order gradient. To balance this, λ-FAdaMax incorporates a decay factor that allows the second moment to use a first-order derivative and gradually transition to a fractional-order derivative as iterations progress. Experimental results across three different deep neural network (DNN) training scenarios demonstrated that λ -FAdaMax accelerates convergence and achieves higher convergence precision.

These articles all suggest that fractional order gradients can improve the speed of convergence and the precision of the minima found.

III. Fractional order gradient descent method

This work developed a new neural network optimizer, called FracGradient, based on Algorithm 3 of the article [1]. The main distinction between this work and others in this area is that the fractional order α is chosen via a function of the norm of the gradient of the loss function $||\nabla J(\Theta)||$. The implemented algorithms support a family of functions, but in this report, we considered the following function:

$$\alpha(N) = 1 - \frac{2}{\pi}\arctan(\beta N) \tag{1}$$

where is N = $||\nabla J(\Theta)||$. We infer that when $N \approx 0$ one has $\alpha(N) \approx 1$, and when $N \gg 0$ one has $\alpha(N) \approx 0$. That is to say, when $||\nabla J(\Theta)||$ is near 0 (when our algorithm has converged), the fractional order of the gradient considered is gonna be near 1, meaning that we fall back to classical derivatives, but when $||\nabla J(\Theta)|| > 0$ the fractional order of the gradient considered will be between 0 and 1.

The central idea is to allow greater flexibility in the learning process by adjusting the influence of the fractional exponent α, thereby controlling the intensity of the updates.

A. Algorithm

The following pseudocode in Algorithm 1 describes the implementation of one of our proposed algorithms, FracGradient.

Algorithm 1: FracGradient

```
1: ▷ Inputs:
2: ▷ Function: $J(\Theta)$
3: ▷ Fixed parameters: $\alpha(N), \beta, \theta, k$
4: for $i \leq k$ do
5:     $f \leftarrow \alpha(||\nabla J(\Theta_{i-1})||)$
6:     $\Theta_{i+1} \leftarrow \Theta_i - \theta\frac{\nabla J(\Theta_{i-1})}{\Gamma(2-f)}|\Theta_i - \Theta_{i-1}|^{1-f}$
7: end
```

- The function $J(\Theta)$ is the loss function of our model that we want to minimize during training.
- The variable Θ is a list of matrices containing the weights of our Neural Network.
- The function $\alpha(N)$ and β define the function that controls the fractional order of the gradient.
- The constant θ is the learning rate.
- The constant k is the number of epochs we want the optimizer to run for.

In essence, this algorithm determines the fractional order of the gradient f. It applies the formula described in [1] to compute ${}^{H}\nabla^{\alpha(N)}_{\Theta^{+}_{i-1}} f(\Theta_i)$, the fractional order gradient, and then updates the weights similarly to the standard gradient descent but with the fractional order gradient. The computation of $J(\Theta)$ is done in the regular manner via backpropagation in the neural network using the standard derivatives.

The algorithm FracGradient V2 described in Algorithm 2 is an extension of the previous algorithm, but the fractional order of the gradient f_n is estimated for each layer of the model, whereas in the last algorithm, this order was fixed for the entire neural network.

Algorithm 2: FracGradient V2

```
1: ▷ Inputs:
```

```
2:  ▷ **Function**: $J(\Theta)$
3:  ▷ **Fixed parameters**: $\alpha(N), \beta, \theta, k$
4:  **for** $i \leq k$ **do**
5:      **for** $n \leq$ num_layers **do**
6:          $f_n \leftarrow \alpha\left(\left|\left|\nabla J\left(\Theta_{i-1}^{(n)}\right)\right|\right|\right)$
7:          $\Theta_{i+1}^{(n)} \leftarrow \Theta_i^{(n)} - \theta \frac{\nabla J\left(\Theta_{i-1}^{(n)}\right)}{\Gamma(2-f_n)} \left|\Theta_i^{(n)} - \Theta_{i-1}^{(n)}\right|^{1-f_n}$
8:      **end**
9:  **end**
```

The main change here is that f_n is computed with the norm of the gradient of each layer $\Theta^{(n)}$, and that order is only applied to the corresponding layer. The idea is that each layer can best adapt its own f_n to improve the algorithm's performance.

The FracGradient V2 Adaptive algorithm described in Algorithm 3 is an extension of the last algorithm, where we update the learning rate θ based on whether in that iteration the cost function $J(\Theta_i)$ was lower than the previous iteration cost function $J(\Theta_{i-1})$.

Algorithm 3: FracGradient V2 Adaptive

```
1:  ▷ **Inputs:**
2:  ▷ **Function**: $J(\Theta)$
3:  ▷ **Fixed parameters**: $\alpha(N), \beta, \theta_i, k, r_d, r_i$
4:  $\theta \leftarrow \theta_i$
5:  **for** $i \leq k$ **do**
6:      **for** $n \leq$ num_layers **do**
7:          $f_n \leftarrow \alpha\left(\left|\left|\nabla J\left(\Theta_{i-1}^{(n)}\right)\right|\right|\right)$
8:          $\Theta_{i+1}^{(n)} \leftarrow \Theta_i^{(n)} - \theta \frac{\nabla J\left(\Theta_{i-1}^{(n)}\right)}{\Gamma(2-f_n)} \left|\Theta_i^{(n)} - \Theta_{i-1}^{(n)}\right|^{1-f_n}$
9:      **end**
10:     **if** $J(\theta_i) > J(\theta_{i-1})$ **then**
11:         $\theta \leftarrow \theta * r_d$
12:     **else**
13:         $\theta \leftarrow \theta * r_i$
14:     **end**
15: **end**
```

The updates of θ are controlled via the hyperparameters r_d and r_i, which decrease or increase the value of θ, respectively. The idea is that the algorithm naturally finds the best θ to maximize convergence speed and make it so the choice of the initial learning rate θ_i doesn't significantly impact the algorithm's output, such as divergence.

B. Implementation

The developed algorithms were implemented in a small Python library available at [11]. This Python library allows for the creation of artificial neural networks and the testing of several different optimisers, but it runs on the CPU.

Furthermore, the FracGradient V2 algorithm was implemented as a custom optimiser in the TensorFlow framework by subclassing the base tf.Keras API. Optimizer. Optimiser class. This approach enabled the seamless integration of the new optimiser with the Keras ecosystem, which allows us to test the optimiser in all compatible model architectures, including convolutional neural networks, and run these models on the GPU.

IV. Experiments

A. Structure of Experiments

To evaluate the performance of the FracGradient optimiser, we conducted three sets of experiments using neural network models for image classification tasks. Tests were performed using the MNIST [12], HappyFace [13], and CIFAR-10 [14] datasets with representative yet straightforward architectures. These datasets were chosen because they have different degrees of complexity that illustrate the behaviour of our algorithms in various scenarios.

The MNIST dataset contains grayscale images of handwritten digits (28x28). It includes 60,000 examples for training and 10,000 for testing, divided equally among 10 classes. All data was normalised to a float from 0 to 1, and the classes were converted to one-hot format.

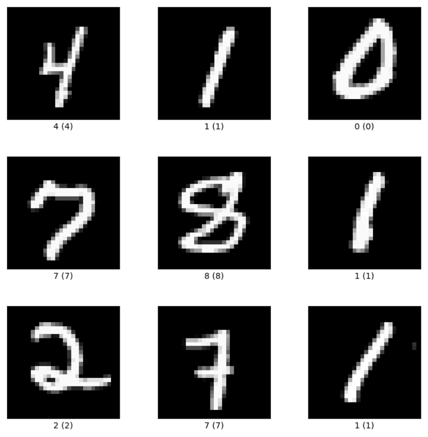

Fig. 1: Sample images from the MNIST dataset.

The HappyFace dataset contains coloured images (64x64) of Happy and Non-Happy facial expressions. With 600 examples for training and 150 for testing. The class distribution is slightly unbalanced. All data was normalised to a float from 0 to 1, and the classes were converted to one-hot format.

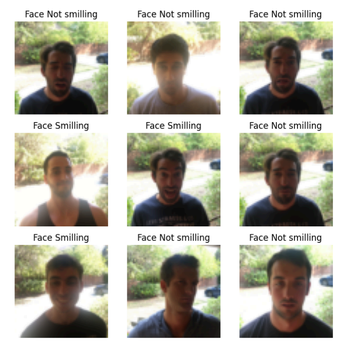

Fig. 2: Sample images from the HappyFace dataset.

CIFAR-10 consists of coloured images (32x32) distributed across 10 classes, with 50,000 images for training and 10,000 for testing, divided equally among 10 classes. All data was normalised to a float from 0 to 1, and the classes were converted to one-hot format.

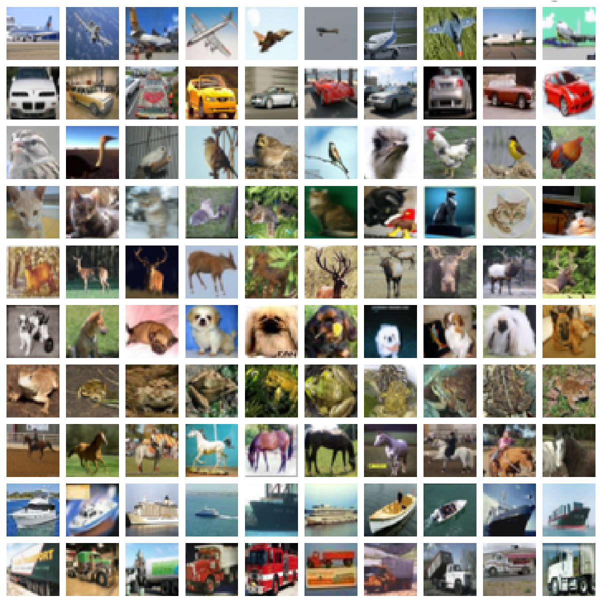

Fig. 3: Sample images from the CIFAR-10 dataset.

A total of 5 experiments were conducted across these datasets.

- For the MNIST dataset, two neural network architectures were tested with slightly different architectures.
- For the HappyFace dataset, one neural network architecture, but with two different sets of activation functions, was tested.
- For the CIFAR-10 dataset, a simple CNN was tested.

For all tests, our algorithms are compared with classical gradient descent, and appropriate hyperparameters, namely the learning rate and β, were chosen by hand.

The comparison of the experiments focuses on the evolution of the cost function (loss) during training, time to finish all epochs, and the final accuracy obtained in the testing data using different optimisers. Accuracy is a suitable metric for these datasets since they are somewhat balanced.

B. Testing Configuration

The tests were conducted in a local Python environment on Windows 11. The host machine running these tests had an AMD Ryzen 7 7800X3D with 32GB of GDDR5 RAM and an NVIDIA RTX 4080 Super GPU. For the experiment on the CIFAR-10 dataset, a WSL environment was used to run the CNN on the GPU.

C. Architectures of Experiments

a) *MLP for MNIST:*

For the MNIST set, an MLP neural network was chosen, which is suitable for small and simple images, such as handwritten digits. The network starts with an input layer of 784 neurons (corresponding to the 28×28 flattened photos). This is followed by a dense layer with 25 or 6 neurons, depending on the test, using sigmoid activation. The output consists of 10 neurons with sigmoid activation, corresponding to the 10 classes. It used the categorical_crossentropy loss function with L2 regularization with a strength of 0.1. One architecture for this experiment is shown in Fig. 4, but only with 32 out of the 784 input neurons. In this case, Θ is a list of 2 matrices with a total of 19885 parameters in the case of 25 neurons in the hidden layer and 4780 parameters in the case of 6 neurons in the hidden layer.

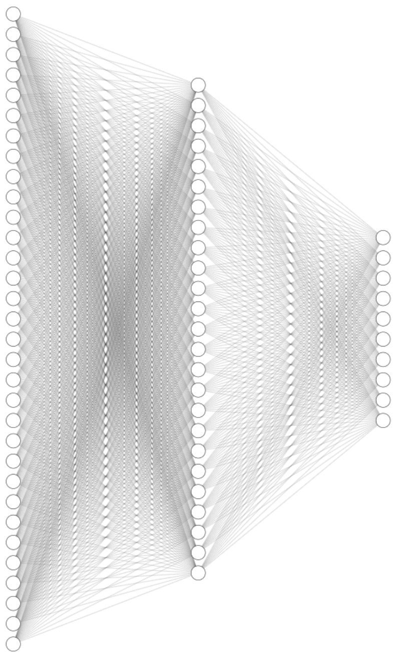

Fig. 4: Architecture of ANN for MNIST dataset with 25 neurons in the hidden layer and the input layer only showing 32 input neurons.

b) *MLP for HappyFace:*

For the HappyFace dataset, an MLP neural network was also chosen. The network starts with an input layer of 4096 neurons (corresponding to the 64×64 flattened photos). This is followed by two dense layers with 64 and 32 neurons, respectively. One experiment used the sigmoid activation function for all layers, while the other used the ReLU activation function and a sigmoid activation function on the output layer. The output consists of 2 neurons, corresponding to the two classes. It used the categorical_crossentropy loss function with L2 regularization with a strength of 0.1. The architecture for this experiment is shown in Fig. 5, but only with 72 out of the 4096 input neurons. In this case, Θ is a list of 3 matrices with 264354 total parameters.

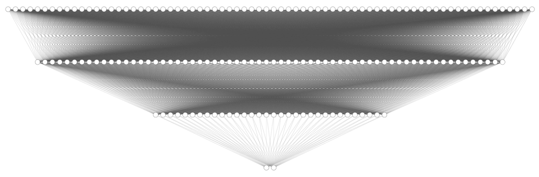

Fig. 5: Architecture of ANN for HappyFace dataset with the input layer only showing 72 input neurons.

c) *CNN for CIFAR-10:*

For the CIFAR-10, a CNN was chosen due to its increased complexity compared to the other two datasets.

The network architecture is as follows:

1) Input and Normalization: The model starts by processing input images of shape (32,32,3). This input first passes through a BatchNormalization layer, which normalizes the activations of the preceding layer.
2) Convolutional Blocks: The normalized input then goes through a series of four convolutional blocks, each consisting of a Conv2D layer, followed by a MaxPooling2D layer, and a Dropout layer:
   - Block 1: A Conv2D layer with 32 filters (implicitly using a 3x3 kernel) processes the 32x32x3 input, maintaining the spatial dimensions at 32x32 but increasing the depth to 32 feature maps. This is followed by a MaxPooling2D layer (implicitly with a 2x2 pool size) that downsamples the feature maps to 16x16, reducing computational load. A Dropout layer is then applied to prevent overfitting.
   - Block 2: Another Conv2D layer with 64 filters processes the 16x16x32 feature maps, outputting 16x16x64 feature maps. This is followed by MaxPooling2D, reducing the dimensions to 8x8x64, and another Dropout layer.
   - Block 3: A Conv2D layer with 128 filters processes the 8x8x64 feature maps, outputting 8x8x128. MaxPooling2D further reduces dimensions to 4x4x128, followed by Dropout.
   - Block 4: The final convolutional block uses a Conv2D layer with 256 filters, transforming the 4x4x128 feature maps into 4x4x256. MaxPooling2D reduces this to 2x2x256, and a final Dropout layer is applied.
3) The 2x2x256 feature maps are then transformed into a one-dimensional vector by a Flatten layer, resulting in a vector of 1024 features (2×2×256=1024). This flattened output is fed into a Dense (fully connected) layer with 512 neurons. Another Dropout layer is applied after this dense layer.
4) Finally, an output Dense layer with 10 neurons is used with softmax activation.

The total number of trainable parameters in this model is 918,358, corresponding to a model size of approximately 3.50 MB and a diagram of the architecture is shown in Fig. 6.

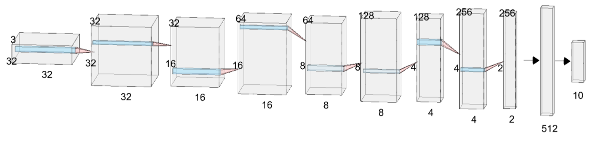

Fig. 6: Architecture of CNN for CIFAR-10 dataset

V. Results

A. MNIST

a) *Experiment 1:*

The plot Fig. 7 illustrates the performance of various gradient descent optimization algorithms in minimizing the cost function $J(\Theta)$ over 3000 iterations for the architecture with 25 hidden layers. Five methods are compared: standard Gradient Descent, Adaptive Learning Rate, FracGradient, FracGradient V2, and FracGradient V2 Adaptive. The Adaptive Learning Rate is an enhancement of Gradient Descent that adjusts the step size dynamically during training, allowing for faster and more stable convergence. Among the methods evaluated, the Adaptive Learning Rate approach demonstrated the fastest convergence and achieved the lowest final cost value. The standard Gradient Descent and FracGradient methods performed worse than their Adaptive learning rate counterpart. FracGradient V2 showed moderate improvement over the original FracGradient, but still lagged behind the adaptive methods. The FracGradient V2 Adaptive method achieved improved convergence rates compared to its non-adaptive counterparts; however, it displayed noticeable oscillations and instability, likely due to fluctuations in the adaptive learning rate.

In summary, adaptive learning strategies, particularly the Adaptive Learning Rate method, offer superior performance in minimizing the cost function in this experiment, and FracGradient algorithms offered no performance gains.

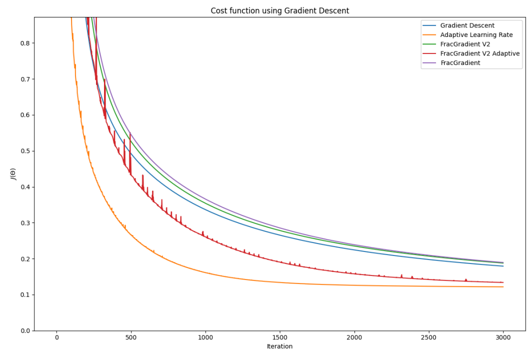

Fig. 7: Cost function $J(\Theta)$ over iterations for different optimisers.

This performance trend is further supported by the accompanying Table I, which compares the optimizers in terms of training and test accuracy, final loss, and execution time. All methods achieved high training accuracy (99–100%) and consistent test accuracy (93%), indicating that model generalization remained stable across optimizers.

Notably, Adaptive Learning Rate achieved the lowest final loss (0.1216). Standard Gradient Descent slightly outperformed FracGradient V2 regarding final loss (0.1877 vs. 0.1791). The same happens in their adaptive counterparts, where FracGradient V2 Adaptive performed marginally worse than the standard adaptive learning rate regarding final loss (0.1216 vs. 0.1336).

Overall, while all methods yielded comparable accuracy, adaptive methods—particularly Adaptive Learning Rate—delivered better optimization in terms of loss minimization. The FracGradient methods were also slower by around 3 seconds, a 5.5% increase in time compared to the Standard Gradient Descent.

Optimiser	**Train Accuracy(%)**	**Test Accuracy(%)**	**Final Loss**	**Time(s)**
Gradient Descent	99	93	0.1791	53.78
Adaptive Learning Rate	100	93	0.1216	53.75
FracGradient	99	93	0.1893	57.52
FracGradient V2	99	93	0.1877	56.72
FracGradient V2 Adaptive	100	93	0.1336	56.73

For the second experiment, with only 6 neurons in the hidden layer. As observed in the plot Fig. 8, Adaptive Learning Rate continues to demonstrate superior convergence behavior, consistently achieving the lowest cost values. FracGradient V2 and FracGradient V2 Adaptive outperform standard Gradient Descent, with faster cost reduction and better final performance. Notably, FracGradient struggles significantly in this scenario, showing delayed and unstable convergence, particularly evident in the cost function curve's late descent and oscillatory behavior.

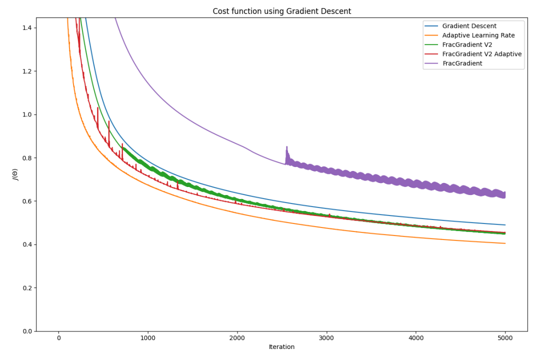

Fig. 8: Cost function $J(\Theta)$ over iterations for different optimisers.

The quantitative results from the Table II show these trends:

- FracGradient V2 attains the best test accuracy (87%) and a lower final loss (0.4529) than Gradient Descent and FracGradient.
- FracGradient V2 Adaptive shows the highest training accuracy (94%) and competitive performance on test accuracy (86%) and final loss (0.4545)
- Despite the final loss function value being significantly higher than all other methods, the test and train accuracy are comparable to Adaptive Learning Rate.

These findings suggest that the FracGradients could benefit the test accuracy for more "complex" tasks, such as this one, where the models struggle to achieve higher than 90% accuracy, despite their final loss values being slightly worse than the Adaptive Learning Rate. To note that the time difference now drops to around 2.8%.

Optimiser	**Train Accuracy(%)**	**Test Accuracy(%)**	**Final Loss**	**Time(s)**
Gradient Descent	91	84	0.4896	84.44
Adaptive Learning Rate	92	86	0.4045	84.434
FracGradient	92	86	0.6406	87.79
FracGradient V2	93	87	0.4529	86.35
FracGradient V2 Adaptive	94	86	0.4545	86.62

B. *HappyFace*

In this experiment, the optimizers were evaluated on the Happy Face dataset using a neural network architecture composed entirely of sigmoid activation functions. This setup, though intentionally suboptimal, offers insight into optimizer behavior under less-than-ideal learning conditions.

The cost function plot Fig. 9 clearly shows that FracGradient V2 Adaptive dramatically outperforms all other methods. It achieves rapid and substantial cost reduction within the first few hundred iterations, whereas Gradient Descent, Adaptive Learning Rate, and FracGradient exhibit almost no learning progress over 1000 iterations, remaining nearly flat.

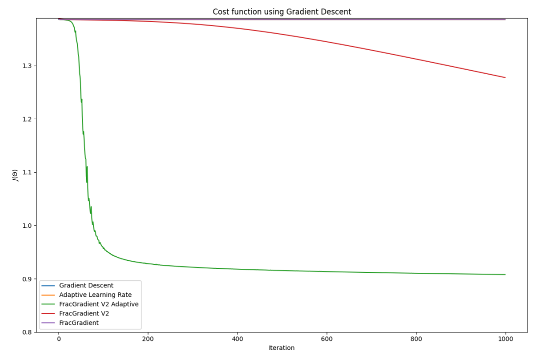

Fig. 9: Cost function $J(\Theta)$ over iterations for different optimisers.

The accompanying Table III confirms this observation:

- The baseline methods—Gradient Descent, Adaptive Learning Rate, and FracGradient—all stagnated at 50% training and 44% test accuracy, with no improvement in loss (1.386), which is consistent with random guessing for a binary classification task using cross-entropy loss.
- FracGradient V2 begins to show signs of effective learning with improved training (74%) and test accuracy (63%), and a slightly reduced loss (1.277).
- FracGradient V2 Adaptive clearly stands out, achieving the highest training (78%) and test accuracy (65%) and the lowest loss (0.9078), demonstrating not only better optimization but also enhanced generalization even with improper activation functions.

These results highlight the robustness of FracGradient V2 Adaptive, which proves capable of overcoming challenges posed by less expressive activation functions. In contrast, traditional gradient methods fail to make meaningful progress, underscoring the advantage of adaptive fractional order approaches in more constrained or ill-conditioned learning scenarios.

Optimiser	**Train Accuracy(%)**	**Test Accuracy(%)**	**Final Loss**	**Time(s)**
Gradient Descent	50	44	1.386	63.95
Adaptive Learning Rate	50	44	1.386	64.82
FracGradient	50	44	1.386	70.31
FracGradient V2	74	63	1.277	70.07
FracGradient V2 Adaptive	78	65	0.9078	70.28

For the fourth experiment, we considered a more proper set of activation functions, namely ReLU and one sigmoid at the output layer. The cost function plot on Fig. 10 shows that among all methods, FracGradient V2 achieved the best overall performance. The cost function curve for this method was smooth and steadily decreasing, indicating consistent progress without major oscillations or divergence. The Adaptive Learning Rate optimiser also performed well, while it exhibited occasional spikes during training, its initial convergence was the fastest among all methods.

Gradient Descent showed steady but slower convergence; however, it suffered from late-stage instability, marked by sharp cost spikes after around 1300 iterations, likely due to the fixed step size overshooting near convergence.

In contrast, FracGradient V2 Adaptive displayed significant fluctuations throughout training, likely caused by unstable adaptive updates.

The original FracGradient method was the least effective. Its training was dominated by persistent high-frequency oscillations, which indicate a failure to converge to an optimal solution.

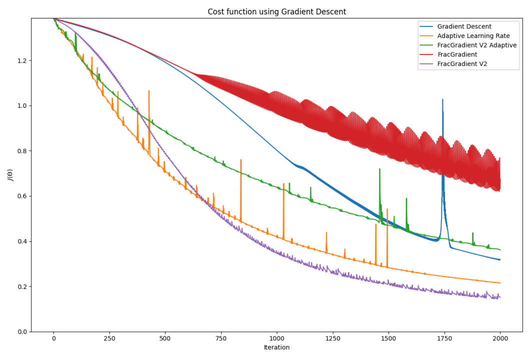

Fig. 10: Cost function $J(\Theta)$ over iterations for different optimisers.

The results presented in Table IV quantitatively reinforce the insights drawn from the cost function analysis. Among all optimisers, FracGradient V2 achieved the lowest final loss (0.1514) while maintaining 99% training accuracy and 95% test accuracy, indicating superior optimisation performance with strong generalisation. The Adaptive Learning Rate method also performed competitively, reaching 99% training accuracy, 95% test accuracy, and a lower final loss (0.2149) than Gradient Descent, all while requiring nearly the same computational time. Gradient Descent, while still effective, showed slightly inferior performance with a final loss of 0.3191 and 97% training accuracy, though it preserved the same 95% test accuracy.

FracGradient V2 Adaptive achieved a comparable 97% training accuracy and 95% test accuracy, but with a noticeably higher final loss (0.3615) and increased computational cost, reflecting less efficient convergence likely due to instability in its adaptive update mechanism. Finally, FracGradient demonstrated the weakest training performance, with only 90% accuracy and the highest final loss (0.6750). Overall, these results show that with proper activations, traditional and fractional optimisers alike can achieve strong generalisation, but FracGradient V2 consistently offers improvements in loss minimisation.

Optimiser	**Train Accuracy(%)**	**Test Accuracy(%)**	**Final Loss**	**Time(s)**
Gradient Descent	97	95	0.3191	127.11
Adaptive Learning Rate	99	95	0.2149	127.41
FracGradient	90	85	0.6750	168.84
FracGradient V2	99	95	0.1514	168.47
FracGradient V2 Adaptive	97	95	0.3615	169.76

In this case, the computational time increases by 32% when comparing standard methods and the FracGradient algorithms. This result is supported by Fig. 11, which shows that until around 1300 epochs, the training does fewer epochs per time than after that point.

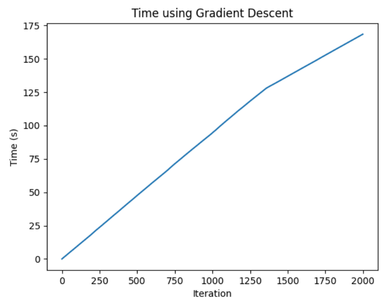

Fig. 11: Cumulative time per epoch of FracGradient V2

C. *CIFAR-10*

For the final experiment, we trained a CNN on the CIFAR-10 dataset. The first set of graphs Fig. 12, corresponding to the ADAM optimizer, displays a rapid initial increase in both training and validation accuracy, reaching a plateau around 0.75-0.78 for validation accuracy after approximately 20-30 epochs. Concurrently, the training and validation loss curves show a sharp initial decrease, stabilising around 1.2. Notably, the validation accuracy and loss curves exhibit more fluctuation than the training curves.

For the Stochastic Gradient Descent (SGD) optimiser, the second set of graphs on Fig. 13 indicate a steadier, more gradual improvement in training accuracy, which continues to rise towards 0.9 by epoch 100. The validation accuracy, however, plateaus earlier, around 0.78-0.80, and shows a noticeable gap from the training accuracy, indicative of potential overfitting as the training progresses. The loss curves reflect this trend, with training loss continuously decreasing while validation loss flattens and even shows some minor increases towards the later epochs.

Finally, the graphs representing the FracGradient V2 optimiser on Fig. 14 show a learning trajectory similar to SGD. The training accuracy increases robustly, approaching 0.9 by the end of 100 epochs, similar to SGD. The validation accuracy demonstrates a consistent upward trend, reaching approximately 0.79-0.80. The gap between training and validation accuracy is present. The loss curves for FracGradient display a smooth and continuous decrease for both training and validation, with the validation loss showing a relatively stable and low value compared to the other optimisers, particularly in the later epochs. This suggests that FracGradient might offer a more stable convergence and potentially better generalisation capabilities on the CIFAR-10 dataset, maintaining a steady improvement in accuracy while consistently reducing loss throughout the training process.

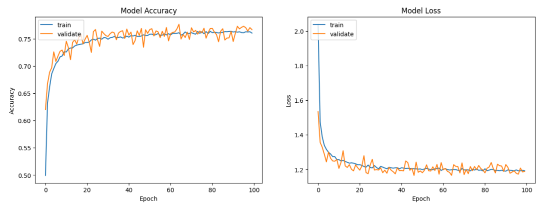

Fig. 12: Cost function $J(\Theta)$ over iterations for ADAM optimiser.

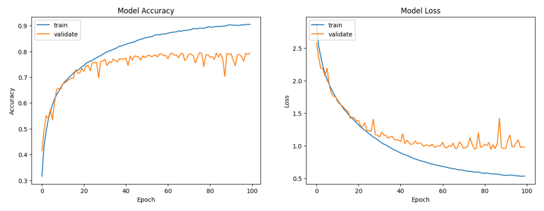

Fig. 13: Cost function $J(\Theta)$ over iterations for SGD optimiser.

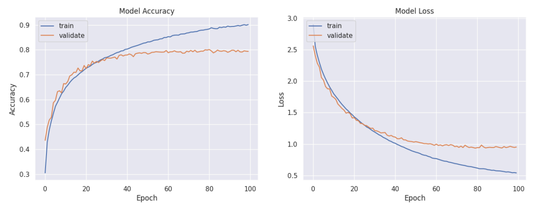

Fig. 14: Cost function $J(\Theta)$ over iterations for FracGradient V2 optimiser.

The Table V concisely summarises the performance metrics for three different optimisers: SGD, ADAM, and FracGradient V2, used on a CNN on CIFAR-10. It clearly shows that SGD and FracGradient V2 achieve high training accuracies (around 90%), with very similar and competitive test accuracies (around 79%). Their final loss values are also relatively low and comparable. In contrast, ADAM exhibits a lower training accuracy (77%) and a slightly lower test accuracy (76.6%), accompanied by a significantly higher final loss. This suggests that while ADAM might converge quickly, SGD and especially FracGradient V2 appear to achieve better final accuracy and loss for this particular setup.

Optimiser	**Train Accuracy(%)**	**Test Accuracy(%)**	**Final Loss**
SGD	90.7	79.3	0.5329
ADAM	77.0	76.6	1.1700
FracGradient V2	90.2	79.4	0.5380

VI. Discussion

The experimental results reveal a nuanced and context-dependent performance of the proposed FracGradient optimization methods. The optimizers did not universally outperform traditional approaches, but demonstrated significant advantages in specific, challenging scenarios, suggesting they are a valuable tool for more complex optimization tasks.

The most compelling evidence for the benefit of fractional gradients emerged from experiments designed to be intentionally difficult. In the HappyFace experiment, using a suboptimal architecture with only sigmoid activation functions, traditional methods like Gradient Descent and Adaptive Learning Rate completely failed to learn, with the loss function stagnating as if guessing randomly. In stark contrast, FracGradient V2 Adaptive was able to successfully navigate this challenging loss landscape, achieving significant cost reduction and resulting in a model with 65% test accuracy. This highlights the robustness of the adaptive fractional order approach, suggesting it is more capable of overcoming the difficulties posed by less expressive activation functions or ill-conditioned problems.

Further evidence of their utility in complex tasks was observed in the CIFAR-10 experiment. When training a Convolutional Neural Network (CNN), FracGradient V2 proved to be a top-tier optimizer. It achieved a final test accuracy of 79.4%, which was competitive with SGD and notably better than ADAM. The training progression showed stable and continuous improvement in both accuracy and loss, suggesting that the fractional method provides excellent generalization capabilities on complex, high-dimensional datasets like CIFAR-10.

However, on the simpler and well-behaved MNIST dataset, the advantages of fractional methods were not apparent. The Adaptive Learning Rate optimizer consistently achieved the fastest convergence and the lowest final loss. While the FracGradient methods yielded comparable final accuracies, they offered no performance gains in terms of convergence speed or loss minimization and, in fact, introduced a slight computational overhead. This indicates that for straightforward optimization problems where standard gradients are effective, the additional complexity of the fractional approach may not be warranted.

Across the experiments, the FracGradient V2 method, which applies a distinct fractional order to each layer, consistently outperformed the original FracGradient algorithm. The original method often struggled with instability and was the least effective optimizer in several tests. This superiority suggests that allowing individual layers to adapt their update intensity based on their local gradient norm is a critical and effective design choice. While the FracGradient V2 Adaptive variant showed promise in the most difficult scenario, it often introduced significant oscillations and instability in other experiments, making the non-adaptive FracGradient V2 a more reliable choice for achieving smooth convergence.

A consistent trade-off with the fractional gradient methods was their increased computational cost. The experiments recorded a 5.5% to 32% increase in training time compared to standard gradient descent. While this is a notable drawback, the performance gains in complex, non-ideal, or large-scale scenarios could justify the additional computational expense. The findings collectively position FracGradient V2 not as a universal replacement for optimizers like Adam or SGD, but as a powerful alternative for tackling complex optimization problems where traditional methods may falter.

The foundational update rule used in this work is adapted from the ψ-Hilfer fractional derivative and shares its core formulation with the simplified fractional order gradient method proposed by Sheng et al. [6]. A key architectural parallel is the strategy of applying fractional updates to the parameters within each layer while preserving the standard integer-order chain rule for backpropagation between layers. While Sheng et al. [6] demonstrated faster convergence and improved accuracy on the MNIST dataset with a fixed fractional order α, the results of this report present a more nuanced view. On MNIST, FracGradient methods were competitive but did not always surpass a simple adaptive learning rate optimizer. The primary distinction and novelty of this report's method is the dynamic adjustment of the fractional order via the function $\alpha(N)$, which depends on the gradient norm. This can be seen as an extension of the work by Sheng et al. [6], designed to automate the optimization of the fractional order, which they identified as a critical parameter for performance.

The success of FracGradient V2 in challenging scenarios strongly aligns with the findings of Shin et al. [8], who demonstrated that their Caputo fractional GD (CfGD) and Adam (CfAdam) variants could accelerate convergence, particularly in ill-conditioned problems. The dramatic performance improvement seen in the intentionally suboptimal HappyFace experiment, where traditional methods failed, mirrors the ability of CfGD to mitigate dependence on the problem's condition number, as highlighted by Shin et al. [8]. Methodologically, their CfGD-v2 incorporates a memory effect by using a previous iterate as an adaptive integral terminal, a concept that is functionally similar to the use of the $|\Theta_i - \Theta_{i-1}|^{1-f}$ term in this report's algorithms. Therefore, the findings of this study provide further evidence that fractional gradient methods with memory are a robust alternative, confirming their potential to outperform standard optimizers in complex, non-convex landscapes.

VII. Conclusion

This project set out to investigate the potential of a new optimisation method based on fractional gradients with a trained approach on the fractional order, with the primary objective of determining if this approach could offer advantages in performance and convergence for neural networks. Through the implementation and evaluation of the FracGradient, FracGradient V2, and FracGradient V2 Adaptive optimisers, this report has demonstrated that fractional-order methods hold significant promise, particularly for complex and challenging optimisation tasks.

The findings indicate that the effectiveness of fractional gradient methods is highly task-dependent. On straightforward problems like the MNIST classification task, the proposed optimisers did not offer a distinct advantage over well-established adaptive learning rate methods. However, their true potential was revealed in more demanding scenarios.

The research also highlighted that a layer-wise application of the fractional order (FracGradient V2) is more stable and effective than a global order. While these benefits come with a trade-off of increased computational time, the performance gains in challenging contexts can justify the cost.

In conclusion, this work successfully documents the implementation of a novel class of fractional-order optimisers and validates their effectiveness. While not a universal replacement for all gradient-based methods, the FracGradient V2 algorithm, in particular, stands out as a powerful and robust alternative for navigating the complex and irregular error landscapes, thereby contributing a valuable perspective to the ongoing exploration of advanced optimisation techniques.

A. Future Work

a) *Changing $\alpha(N)$ and rescaling Θ:*

The original paper [1] outlines functions for $\alpha(N)$ that could have an impact on the results of this report. Furthermore, it also allows rescaling Θ through a function ψ that can be used to create a more general algorithm using other types of fractional derivatives, not only Caputo derivatives, in the case when $\psi(x) = x$

b) *Stabilizing β:*

The hyperparameter is dependent on the size of the norm of the gradient; as such, on smaller or bigger matrices of parameters, its effect is different. It could also be studied whether or not this is desirable, or the impact of β can be regularised for any size of $\Theta^{(n)}$.

c) *Testing more architectures:*

Right now, the optimisers were only tested on the three datasets, so these results might not generalise to other datasets and different architectures used in machine learning.

d) *Different approach for learning rate:*

The adaptive learning rate, in some cases, didn't work well with fractional order derivatives. Different approaches, such as learning rate scheduling, could be explored.

e) *Optimising the hyperparameters and Statistical significance:*

As it stands, the hypermeters were chosen by hand, which could introduce bias. A more rigorous approach, such as a systematic grid search or random search for optimal hyperparameters for each optimizer independently, would provide a much more objective and credible comparison. To make a stronger claim, the experiments should be run multiple times with different random seeds, and the results should be reported with means and standard deviations.

References

[1] N. Vieira, M. M. Rodrigues, and M. Ferreira, "Fractional gradient methods via \psi-Hilfer derivative", *Fractal and Fractional*, vol. 7, no. 3, p. 275, 2023.

[2] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016, [Online]. Available: https://arxiv.org/abs/1609.04747

[3] I. Podlubny, *Fractional Differential Equations*. San Diego, CA, USA: Academic Press, 1999.

[4] C. Li and F. Zeng, *Numerical Methods for Fractional Calculus*. Boca Raton, FL, USA: CRC Press, 2015.

[5] V. E. Tarasov, *Fractional Dynamics: Applications of Fractional Calculus to Dynamics of Particles, Fields and Media*. Berlin, Germany: Springer, 2011.

[6] D. Sheng, Y. Wei, Y. Chen, and Y. Wang, "Convolutional neural networks with fractional order gradient method," *Neurocomputing*, vol. 408, pp. 42–50, 2020.

[7] W. Lou, W. Gao, X. Han, and Y. Zhang, "Variable Order Fractional Gradient Descent Method and Its Application in Neural Networks Optimization," in *2022 China Control and Decision Conference (CCDC)*, 2022. doi: 10.1109/CCDC553562022.10013456.

[8] Y. Shin, J. Darbon, and G. E. Karniadakis, "Accelerating gradient descent and Adam via fractional gradients," *Neural Networks*, vol. 161, pp. 185–201, 2023.

[9] P. Harjule, R. Sharma, and R. Kumar, "Fractional-order gradient approach for optimizing neural networks: A theoretical and empirical analysis," *Chaos, Solitons and Fractals*, vol. 192, p. 116009, 2025, doi: 10.1016/j.chaos.2025.116009.

[10] G. Chen and Z. Xu, "\lambda-FAdaMax: A novel fractional-order gradient descent method with decaying second moment for neural network training", *Expert Systems With Applications*, vol. 279, p. 127156, 2025.

[11] DiogoMMarto, "FracGradient." [Online]. Available: https://github.com/DiogoMMarto/FracGradient

[12] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST Database of Handwritten Digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[13] A. Motwani, "HappyFace Dataset." [Online]. Available: https://www.kaggle.com/datasets/ashishmotwani/happyface

[14] A. Krizhevsky, "CIFAR-10 Dataset: Learning Multiple Layers of Features from Tiny Images," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html