applied sciences

MDPI

Article

Fractional Derivative Gradient-Based Optimizers for Neural Networks and Human Activity Recognition

Oscar Herrera-Alcántara

Departamento de Sistemas, Universidad Autónoma Metropolitana, Mexico City 02200, Mexico; oha@azc.uam.mx

Abstract: In this paper, fractional calculus principles are considered to implement fractional derivative gradient optimizers for the Tensorflow backend. The performance of these fractional derivative optimizers is compared with that of other well-known ones. Our experiments consider some human activity recognition (HAR) datasets, and the results show that there is a subtle difference between the performance of the proposed method and other existing ones. The main conclusion is that fractional derivative gradient descent optimizers could help to improve the performance of training and validation tasks and opens the possibility to include more fractional calculus concepts to neural networks applied to HAR.

Keywords: fractional derivative; gradient descent optimizer; human activity recognition

Citation: Herrera-Alcántara, O. Fractional Derivative Gradient-Based Optimizers for Neural Networks and Human Activity Recognition. *Appl. Sci.* **2022**, *12*, 9264. https://doi.org/10.3390/app12189264

Academic Editor: Valentina E. Balas

Received: 3 August 2022
Accepted: 12 September 2022
Published: 15 September 2022

1. Introduction

In the context of machine learning, neural networks are one of the most popular and efficient techniques to model data, and gradient descent methods are widely used to optimize them. The fundamental gradient descent optimizer considers a factor with an opposite direction to the gradient of the objective function. Other optimizers consider momentum and velocity analogies to improve the training convergence and the generalization capacity.

Effectively, starting with the basic update rule of gradient descent optimizers, the fundamental factor updates the free parameters in the opposite direction of the gradient g_t on the approximation error surface, and the learning rate η modulates the feedback to move forward to obtain a minimum [1].

A batched version vanilla gradient descent (GD) updates the parameters considering all the training samples, but it is impractical for large datasets. The GD update formula is

$$\Delta\theta_t = -\eta g_t. \tag{1}$$

Alternatively, a stochastic gradient descent (SGD) version updates the parameters for each i-th training sample. Hence, the SGD update formula is

$$\Delta\theta_{t,i} = -\eta g_{t,i} \tag{2}$$

and although it could introduce fluctuations, on one hand, it can be useful to explore the optimization space, but on the other hand, it can introduce unnecessary variance in the parameter updates, and it makes the learning rate a critical factor. Considering this, a mixed version of minibatch gradient descent proposes to split the dataset in subsets to deal with this tradeoff [2].

Adagrad [3] is the other evolved version that considers adaptation of the learning rate based on the memory of the gradients and aims to give helpful feedback for sparsed features of input data. Adagrad computes a historical diagonal matrix $G_{t,ii}$, accumulating the sum of squares of the gradients to modify the adjustment of each parameter θ_i which aims to

deal with the disparity of frequent/infrequent features of training samples. The Adagrad update formula is

$$\Delta\theta_{t,i} = -\frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i} \tag{3}$$

that considers $\epsilon > 0$ in the denominator to avoid a zero division.

Adadelta [4] is a variant of Adagrad that aims to lessen the accumulation of square gradients along all the time for G_{ii}, and instead it defines an average window given $0 \leq \gamma \leq 1$ to ponderate squares of current and previous one gradients according to

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{4}$$

so, it can be conceived as the root mean squared error of the gradients:

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{5}$$

where $\epsilon > 0$ is also included to avoid a division by zero, given that the original update formula for Adadelta is

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t. \tag{6}$$

To preserve the same "unit of measure", the learning rate η is replaced by the RMS of parameter updates in this way:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} = \sqrt{\gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 + \epsilon} \tag{7}$$

up to $t-1$ since $\Delta\theta$ is still being calculated. Therefore, the Adadelta update rule is

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t. \tag{8}$$

RMSProp [5] is considered an extension of Adagrad that maintains a moving average square of the gradients instead of using all the historical, and divides the gradient by the root mean square of that average. In this sense, it has a great similarity with Equation (6) presented as the former Adadelta rule.

Other optimizers consider momentum (update memory) based on the update of the previous iteration, analogous to the physical concept of particle inertia, so when "the ball moves" in the same direction from the current to the next update, it accelerates the convergence, and it opposes when it changes directions, providing more stability and better convergence.

Adam [6] mixes the average of past gradients m_t and past squared gradients v_t together with a momentum approach, in such a case that $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$ contains a previous memory value followed by a second term based on the gradient. Similarly, $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$ first involves a memory update term followed by a squared gradient term. Since m_t and v_t are initialized to zero, and to avoid zero-bias tendency, other formulas are obtained for the first momentum bias-corrected $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$ and for the second momentum bias-corrected $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$. The updated Adam formula is

$$\Delta\theta_{t,i} = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{9}$$

that also considers $\epsilon > 0$ in the denominator to avoid a zero division.

Some other gradient descent optimizers have been proposed, but now just only one more is discussed, the AdamP optimizer [7]. AdamP is a variant of Adam that appeals to weight normalization and effective automatic step size adjustment over time to stabilize the overall training procedure that improves the generalization. The normalization considers

projections in the weight space via the projection operator $\Pi_\theta(x) := x - (\theta \cdot x)\theta$ applied to the momentum update $m_t = \beta m_{t-1} + g_t$, so the AdamP rule is $\Delta\theta_t = -\eta q_t$ where

$$q_t = \begin{cases} \Pi_{\theta_t}(m_t), & \text{if } cos(\theta_t, g_t) \leq \frac{\delta}{dim(\theta)} \\ m_t, & \text{otherwise} \end{cases} \tag{10}$$

for $\delta > 0$, and where $cos(a, b)$ is the cosine similarity between two vectors.

Table 1 summarizes these previously described optimizers. In addition, it also includes a version of SGD with γ momentum, as well as SGDP [7] that includes weight normalizations via projections based on SGD, analogous to how AdamP is built based on Adam. The first column indicates the name, the second column the update rule, and the third column a comment. It is emphasized that the update rule for some optimizers, such as Adagrad, Adadelta and Adam, involves $\epsilon > 0$ to avoid a division by zero.

Table 1. Update rules for several gradient descent optimizers.

Name	Update Rule	Comment for the Update
GD	$\Delta\theta_t = -\eta g_t$	It is opposed to the gradient.
SGD	$\Delta\theta_{t,i} = -\eta g_{t,i}$	It is opposed to the gradient for each training sample.
Adagrad	$\Delta\theta_{t,i} = -\frac{\eta}{\sqrt{G_{t,ii}+\epsilon}} g_{t,i}$	It is opposed to the gradient with adaptive decreasing learning rate for each sample.
Adadelta	$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$	It is opposed to the gradient with adaptive learning rate for each sample.
RMSProp	$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$	It is opposed to the gradient, and divides η using the RMS of the average square of windowed gradients.
SGD with Momentum γ	$\Delta\theta_{t,i} = -\gamma\Delta\theta_{t-1} - \eta g_{t,i}$	It uses one-slot memory of parameter updates and direction opposed to the gradient for each training sample.
Adam	$\Delta\theta_{t,i} = -\frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$	It is opposed to the gradient and combines average of past gradients m_t as well as average of past squared gradients v_t.
AdamP	$\Delta\theta_t = -\eta\Pi_{\theta_t}(m_t)$ or $-\eta m_t$	It is opposed to the gradient and also considers weight normalization via the projection of m_t.
SGDP	$\Delta\theta_{t,i} = -\eta\Pi_{\theta_{t,i}}(\gamma)$ or $-\eta\gamma$	It is opposed to the gradient and also considers weight normalization via the projection of γ.

All previous optimizers consider the gradient g_t as the cornerstone update factor that comes from a first-order derivative of the objective function. The purpose of this work is to present optimizers that introduce a fractional derivative gradient in the update rule, as well as an implementation for the Tensorflow backend. This proposal is mainly based on the fractional differential calculus theory [8–11] and on previous works [12,13].

Fractional calculus is not a novel topic [14] but it has recently taken relevance in several fields, including linear viscoelasticity [15], fractional control [16], partial differential equations [17], signal processing [18], image processing [19], time series prediction [20], and mathematical economics [21], among others, and of course neural networks in the age of deep learning [12,13,22]. Given that neural network architectures have several challenges such as generalization enhancement, gradient vanishing problems, regularization and overfitting, it seems that fractional calculus still has a lot to contribute.

The rest of the paper is organized as follows. In Section 2, details of the proposed fractional derivative gradient update rule are presented. In Section 3, experiments are described to obtain performance comparisons with known optimizers. It allows to support the main conclusion regarding the improvement between the performance of the proposed method and other existing ones. In Section 4, some discussions are presented based on the experiments, and some future work directions are commented on.

2. Materials

In this section, the Caputo fractional derivative definition is reviewed, as well as its relationship with the backpropagation algorithm for neural networks.

2.1. Fractional Derivatives

There is no unified theory for fractional calculus, and evidence of this is that there is no single definition for fractional derivatives. See, for example, the Grünwald–Letnikov, the Riemann–Liouville and the Caputo definitions [10,13,23]. The Caputo fractional derivative, for $a, x \in \mathbb{R}$, $\nu > 0$ and $n = [\nu + 1]$, is defined as

$$ {}_a^C D_x^\nu f(x) = \frac{1}{\Gamma(n-\nu)} \int_a^x (x-y)^{n-\nu-1} f^{(n)}(y) dy \tag{11} $$

and it seems to be the most popular since, in contrast to Grünwald–Letnikov and Riemann–Liouville, the Caputo fractional derivative of Equation (11) is zero for $f(x) = C$, with $C \in \mathbb{R}$, which matches with the integer derivative version [12]. In Equation (11), a $(x-y)^{n-\nu-1}$ kernel can be identified that convolves with $f^{(n)}$. The application and study of other kernels and their properties to define more fractional derivatives is an open research area.

An interesting property of the fractional ν-order derivative operator D_x^ν applied to x^p is that [24]

$$ D_x^\nu x^p = \frac{\Gamma(p+1) x^{p-v}}{\Gamma(p-v+1)} \tag{12} $$

and, in particular for $\nu = \frac{1}{2}$ and $p = 1$, it allows to calculate the $\frac{1}{2}$ derivative of x:

$$ D_x^{\frac{1}{2}} x = \frac{\Gamma(2) x^{\frac{1}{2}}}{\Gamma(\frac{3}{2})}, \tag{13} $$

moreover, if the $v = \frac{1}{2}$ derivative is calculated again with $p = \frac{1}{2}$, i.e., if $D_x^{\frac{1}{2}}$ is applied again to Equation (13),

$$ D_x^{\frac{1}{2}+\frac{1}{2}} x = D_x^{\frac{1}{2}} (D_x^{\frac{1}{2}} x) = D_x^{\frac{1}{2}} \frac{\Gamma(2) x^{\frac{1}{2}}}{\Gamma(\frac{3}{2})} = \frac{\Gamma(2)}{\Gamma(\frac{3}{2})} \frac{\Gamma(\frac{3}{2})}{\Gamma(1)} x^0 = 1, \tag{14} $$

which is consistent with the first-order derivative $D_x^{(1)} x = 1$.

2.2. Backpropagation for Neural Networks

In supervised learning, given an input data set X and the corresponding desired outputs O, the training sample set can be expressed as $\{X^i, O^i\}_{i=1}^N$, where N is the number of samples.

For a neural network with an architecture conformed by a single input layer X, followed by $L = H + 1$ layers that considers H hidden layers and an output layer O with activation functions $\varphi(x)$, the matrix of synaptic weights w_{kj}^l indicates the connection between the neuron k of layer $l+1$ and the neuron j of the current layer, $l \in [1, L-1]$. A special case is for $l = 0$, where the weights connect the input data X with the neurons of the first hidden layer.

The error of neuron k at the output layer is $e_{ki} = a_{ki}^L - o_{ki}$, where subindex i refers to the neural network receiving the i-th input pattern. Consequently, given the i-th training sample, the error E_i of the output layer considering all its n^L neurons is

$$ E_i = \frac{1}{2} \sum_{k=1}^{n^L} e_{ki}^2 = \frac{1}{2} \sum_{k=1}^{n^L} (a_{ki}^L - o_{ki})^2 \tag{15} $$

then, the total error E over all the N training samples is

$$ E = \sum_{i=1}^{N} E_i = \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{n^L} (a_{ki}^L - o_{ki})^2 \tag{16} $$

and the learning process via the backpropagation algorithm aims to minimize E by adjusting the free parameters of the weight matrix.

Essentially, a backpropagation training consists of repeated forward and backward steps. The forward step evaluates progressively the induced local fields V^l, multiplying the inputs I^l of the l-th layer and the corresponding synaptic weights $W^l = w^l_{kj}$. For the first layer, $I^1 = X$, so the induced local field vector at layer l can be expressed as the dot product $V^l = I^l \cdot W^l$ where $l \in [1, L]$.

The output of neuron k at layer l is $a^l_k = \varphi(v^l_k)$, where v^l_k is the k-th local induced field of V^l, and by convention for $l = 0$ the "output vector" a^0 is equal to $I = X$, the input data set. Of course, each activation function φ can be different for each layer.

For the backward step, once the outputs a^L of the L-th layer have been calculated, the local gradients δ^l_k are evaluated, and it allows to obtain the gradient descent updates in reverse order for $l = L, L-1, \ldots, 1$.

Indeed, for the gradient descent optimizer, the weight updates Δw^l_{kj} are given by

$$\Delta w^l_{kj} = -\eta \frac{\partial E_i}{\partial w^l_{kj}} \tag{17}$$

seeking a direction for weight change that reduces the value of E_i [1].

Since the local gradient is

$$\delta^l_k = \frac{\partial E_i}{\partial v_k} \tag{18}$$

and considering that

$$\frac{\partial E_i}{\partial w^l_{kj}} = \frac{\partial E_i}{\partial v_k} \cdot \frac{\partial v_k}{\partial w^l_{kj}} = \frac{\partial E_i}{\partial v_k} \cdot a^{l-1}_j = \delta^l_k a^{l-1}_j \tag{19}$$

then, Δw^l_{kj} can be expressed as

$$\Delta w^l_{kj} = -\eta \cdot \delta^l_k \cdot a^{l-1}_j. \tag{20}$$

At the output layer, δ^L_k involves two factors, the error e_{ki} and the derivative of the activation function as follows:

$$\delta^L_k = e_{ki} \cdot \varphi'(v^L_k) \tag{21}$$

whereas for hidden layer l, the local gradient considers the contribution of errors via the k neurons of the $l+1$ layer, hence

$$\delta^l_j = \varphi'(v_j) \cdot \sum_{k=1}^{n^{l+1}} \delta^{l+1}_k \cdot w^{l+1}_{kj}. \tag{22}$$

To be consistent with the nomenclature of Section 1, let $g_t = \delta^l_k \cdot a^{l-1}_j$, where a^{l-1}_j is the output of neuron j of the previous layer, i.e., an input to the layer l. Additionally, let $\Delta\theta_{t,i} = \Delta w^l_{kj}$ when the i-th training sample is presented to the neural network.

2.3. Fractional Derivative and Gradient Descent

Essentially, the same approach of the gradient descent for the first-order derivatives is applied to the fractional gradient $D^v_{w^l_{kj}} E_i$. In this case, the weight updates are

$$\Delta w^l_{kj} = -\eta D^v_{w^l_{kj}} E_i \tag{23}$$

and the main difference comes when applying the chain rule, as follows:

$$D^{\nu}_{w^l_{kj}} E_i = \frac{\partial E_i}{\partial w^l_{kj}} \cdot D^{\nu}_{w^l_{kj}} w^l_{kj} = \delta^l_k \cdot a^{l-1}_j \cdot \frac{(w^l_{kj})^{1-\nu}}{\Gamma(2-\nu)}, \tag{24}$$

which is identical to that of the integer derivative but multiplied by the fractional factor $\frac{(w^l_{kj})^{1-\nu}}{\Gamma(2-\nu)}$.

Note that the property of Equation (12) for $p = 1$ is applied to obtain the fractional ν-order derivative of w^l_{kj}. Additionally, note that in the case of $\nu = 1$, it is reduced to the already known integer case since the factor $(w^l_{ji})^{1-\nu} = 1$ and $\Gamma(2-\nu) = 1$, then Equation (24) can be conceived as a generalization of the integer gradient descent, for $\nu > 0$.

2.4. Tensorflow Implementation of Fractional Gradient Optimizers

Tensorflow is a platform for machine learning, and it has been widely used for the deep learning community since it provides open-source Python libraries to train and deploy many applications [25]. Tensorflow also includes efficient support for GPU devices, as well as integration with high-level APIs, such as Keras [26]. Tensorflow is available for several operating systems and is also available through Jupyter notebook cloud services, such as Google Colab [27].

The module *tf.optimizers* contains classes for gradient descent optimizers, such as SGD, Adadelta, Adagrad, Adam, among others. For example, the SGD optimizer is located in the Tensorflow–Keras module *tf.keras.optimizers.SGD*, and accepts some parameters, as is shown in the next fragment of code:

```
tf.keras.optimizers.SGD(
        learning_rate=0.01,
        momentum=0.0,  ...
)
```

These parameters have default values, such as $momentum = 0$, that means that the default update rule is $\Delta w = -learning\_rate * gradient$. Given a positive value of momentum, the update rule according to the API documentation is $\Delta w = velocity$ where the "velocity" is defined as $velocity = momentum * velocity - learning\_rate * gradient$. So, the velocity stores a single slot memory value as described in Section 1, and it corresponds to the $\Delta\theta_{t,i}$ factor in the fifth row of Table 1.

Since the main goal is to introduce the fractional factor of Equation (24) to the gradient descent optimizers, a simple and elegant solution is to multiply the current gradient by this factor. However, there are some aspects to be considered. First, note that Equation (24) involves a power $1-\nu$ that will be negative for $\nu > 1$, and consequently, it could produce a division by zero (in the practice, Tensorflow obtains NaN values). A possible solution is to aggregate $\epsilon > 0$, as it was shown in Section 1. However, there is a second consideration; when $1 - \nu = \frac{p}{q}$, and q is even (for example $\nu = \frac{1}{2}$ or $\nu = \frac{3}{4}$), then negative values of w^l_{kj} generate complex values. To deal with these two situations and to preserve real values, w^l_{kj} was replaced by $|w^l_{kj}| + \epsilon$, so the proposed fractional gradient factor f^{ν}_w is

$$f^{\nu}_w := \frac{(|w^l_{kj}| + \epsilon)^{1-\nu}}{\Gamma(2-\nu)}. \tag{25}$$

A strong motivation to replace w^l_{kj} by $|w^l_{kj}| + \epsilon$ is that it allows to have a limit for f^{ν}_w when $\nu \to 1$. In such a case,

$$\lim_{\nu \to 1^-} \frac{(|w^l_{kj}| + \epsilon)^{1-\nu}}{\Gamma(2-\nu)} = \lim_{\nu \to 1^+} \frac{(|w^l_{kj}| + \epsilon)^{1-\nu}}{\Gamma(2-\nu)} = 1, \tag{26}$$

that supports the idea of conceiving Equation (24) as a more general case of the integer gradient descent update rule.

For the Tensorflow implementation, a new class FSGD with fractional gradient was defined based on the SGD optimizer. The *update_step* method was modified as follows:

```
# Definition of tensors: constants and variables. Choose v > 0.
v = 0.5
one_v = 1-v
two_v = 2-v
gamma_2_v = math.gamma(two_v)
epsi = 0.000001
class FSGD(optimizer.Optimizer):
    ...
    def update_step(self, gradient, variable):
        tmp1 = tf.abs(variable) + tf.constant(epsi, tf.float32 )
        tmp2 = tf.constant(one_v, tf.float32 )
        fw = tf.pow(tmp1 , tmp2 )/gamma_2_v
        gradient = tf.multiply(gradient, fw)
```

The same procedure applies to other gradient descent optimizers listed in Table 1, and each fractional version uses the prefix "F". For example, FAdam is the fractional version of Adam, and it was obtained modifying the *_resource_apply_dense* method of the Adam class. The modification includes the next source code:

```
class FAdam(keras.optimizers.Optimizer):
...
    def _resource_apply_dense(self, grad, var, apply_state=None):
        tmp1 = tf.abs(var) + tf.constant(epsi, tf.float32 )
        tmp2 = tf.constant(one_v, tf.float32 )
        fw = tf.pow(tmp1 , tmp2 )/gamma_2_v
        grad = tf.multiply(grad, fw)
```

The source code for AdamP was adapted from [7] and despite it including modifications for the weight normalization via projections, the section of interest to update the gradient is identical to Adam. Thus, the same modifications apply to the fractional version named FAdamP. In a similar manner, it also applies to FSGDP as the fractional version of SGDP.

The source code of all fractional optimizers FSGD, FAdagrad, FAdadelta, FRMSProp, FAdam, FSGDP and FAdamP is available for download.

3. Results

Once the fractional optimizers FSGD, FAdagrad, FAdadelta, FRMSProp, FAdam, FSGDP and FAdamP were implemented, they were compared with their counterparts available in Tensorflow–Keras, as well as with SGDP and AdamP obtained from [7].

The fractional versions with prefix "F" and $\nu = 1.0$ coincide with the original non-fractional versions, since according to Equation (26) they are special cases of the fractional derivatives, and it was comprobated experimentally.

The comparisons were organized in three experiments. The first experiment considers the well-known dataset MNIST [28], whereas Experiments 2 and 3 use the HAR datasets [29,30].

3.1. Experiment 1

Experiment 1 uses MNIST with 10-fold cross-validation, 15 epochs, architecture of 3 dense layers with ReLu, and an output layer with softmax for 10 classes.

Three subexperiments are described below.

3.1.1. Experiment 1.1

It considers a learning rate $\eta = 0.001$ and momentum $\gamma = 0$ for FSGD because the main idea is to evaluate the fundamental effect of the fractional factor f_w^ν. In this case, $\nu = 0.1, 0.2, \ldots, 1.9$ since the experiments show that larger values of ν have worse

performance. Obviously, it considers the case $\nu = 1.0$, whose results match with SGD, and it was corroborated obtaining a correlation of 0.999.

The results of the cross-folding accuracy are shown in Table 2, where the rows correspond to folds 1 to 10. It is possible to appreciate that small values of ν close to zero produce low accuracies, and the worst case is for $\nu = 0.1$ that reports an accuracy of 12.3% at the third fold. Conversely, as ν increases, so does the accuracy, which reaches a maximum and then begins to decrease slowly.

Table 2. Comparison between SGD and FSGD. The case $\nu = 1.0$ matches with SGD.

FSGD ($\nu = 0.1, \ldots, 1.9$)																		
0.1	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**1.7**	**1.8**	**1.9**
18.7	24.7	33.8	43.2	55.9	69.6	81.9	87.2	89.3	**90.9**	92.3	93.2	94.0	94.7	95.2	95.1	**95.3**	94.2	92.2
13.7	17.0	23.3	30.1	44.2	74.5	83.0	87.9	90.4	**92.1**	93.7	94.6	95.1	95.4	95.6	95.6	**95.8**	95.8	94.3
12.3	13.7	18.2	23.0	37.9	73.3	83.1	88.0	90.2	**91.7**	92.8	93.8	94.3	94.8	94.9	95.1	**94.5**	94.5	91.8
19.7	25.0	32.3	42.4	53.8	71.3	83.4	87.8	90.0	**91.2**	92.6	93.7	94.5	95.0	95.3	95.5	**95.4**	94.7	93.3
22.8	26.9	33.7	54.4	68.3	80.5	86.9	89.6	91.0	**92.5**	93.2	93.9	94.5	94.9	95.3	95.5	**95.5**	95.2	93.8
14.8	16.7	18.4	21.0	24.7	61.2	82.1	87.7	89.9	**91.3**	92.2	93.2	94.3	94.7	95.1	95.2	**95.4**	94.4	92.8
19.2	22.9	29.8	42.9	55.5	77.8	85.4	88.0	90.2	**91.5**	92.6	93.4	94.1	94.6	94.9	95.1	**95.0**	94.4	93.0
19.5	26.2	34.5	40.0	45.6	73.1	84.0	88.3	90.0	**91.6**	92.6	93.5	94.3	94.9	95.4	95.6	**95.9**	95.3	94.0
22.7	29.0	36.4	43.2	46.0	49.9	78.2	86.6	89.5	**90.8**	92.0	92.9	93.6	94.2	94.7	95.1	**95.3**	94.5	93.5
25.4	32.9	42.6	51.5	58.6	77.0	85.6	88.4	90.4	**91.6**	92.8	93.7	94.3	94.9	95.4	95.6	**95.5**	95.0	93.3

In Figure 1, the boxplots of all data of Table 2 are shown. In both of them, it is possible to appreciate the optimal performance for $\nu = 1.7$ (the average accuracy is 95.85 and the standard deviation is 0.36). The improvement is about 4% better than SGD ($\nu = 1.0$), and these values are highlighted in bold in Table 2 for comparison purposes.

From the results of Experiment 1, the importance of the fractional gradient factor f_w^ν stands out, since the best performance is achieved for $\nu > 1.0$, instead of the traditional $v = 1$. It is shown that f_w^ν provides additional freedom degree to optimize the neural network parameters.

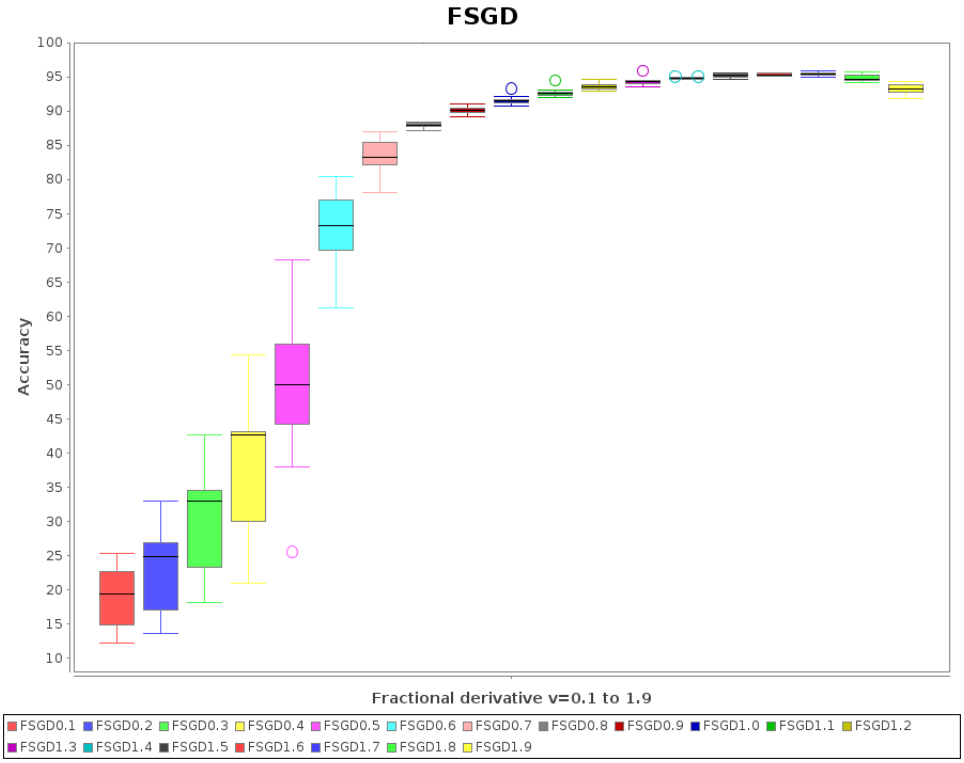

Figure 1. Experiment 1.1: Boxplots for accuracies of FSGD with 10-cross folding and $\nu = 0.1, 0.2, \ldots, 1.9$.

3.1.2. Experiment 1.2

This experiment considers FSGD with the same values for learning rate η and momentum γ. The learning rate is increased 100 times with respect to Experiment 1.1, and then $\eta = \gamma = 0.1$ that aims to have a balance on their contribution to the weight updates. The fractional order varies from $\nu = 0.1$ to 1.9 with step = 0.2, and additionally $\nu = 1.0$, which corresponds to SGD as a special case.

The results of Experiment 1.2 are shown in Table 3 and Figure 2, where it is possible to see (highlighted in bold in Table 3) that cases $\nu = 1.1$ and $\nu = 1.9$ have better performance than others, including the case $\nu = 1.0$ which corresponds to SGD with momentum $\gamma = 0.1$. Although these cases in the last fold (see the last row of columns $\nu = 1.1$ and $\nu = 1.9$) have a slightly smaller value than those of $\nu = 1.0$, the rest of the data show a consistent enhancement over the rest of the folds, as it is illustrated in the boxplots of Figure 2, where the boxplots for $\nu = 1.1$ and $\nu = 1.9$ are better positioned above the case $\nu = 1.0$.

Table 3. Experiment 1.2: Comparison between SGD and FSGD with $\eta = 0.1$ and momentum $\gamma = 0.1$. The case $\nu = 1.0$ corresponds to SGD.

	ν										
Fold	**0.1**	**0.3**	**0.5**	**0.7**	**0.9**	**1.0**	**1.1**	**1.3**	**1.5**	**1.7**	**1.9**
1	99.0	99.0	99.0	98.9	99.1	**98.9**	**99.0**	99.0	98.9	99.0	**99.0**
2	98.6	98.9	98.9	98.9	98.9	**98.9**	**99.0**	99.0	99.0	98.8	**99.1**
3	98.9	98.6	96.1	99.0	98.9	**96.7**	**99.0**	99.0	98.8	98.8	**99.0**
4	98.9	98.8	98.9	98.9	98.8	**98.8**	**98.8**	98.8	98.6	98.9	**98.9**
5	98.7	98.7	98.7	98.8	98.7	**98.7**	**98.8**	98.7	98.7	98.8	**98.8**
6	98.9	98.9	99.1	98.9	98.8	**96.6**	**99.0**	98.8	98.8	98.4	**99.0**
7	99.1	99.0	99.0	99.1	98.9	**99.0**	**99.0**	98.9	99.1	98.7	**99.2**
8	99.3	99.2	98.9	99.2	98.5	**99.1**	**99.2**	99.3	99.1	99.1	**99.1**
9	98.7	98.7	98.8	98.8	98.8	**98.8**	**98.8**	98.8	98.7	98.7	**98.9**
10	98.8	98.9	99.0	98.9	99.0	**99.1**	**98.9**	99.0	98.9	99.0	**98.8**

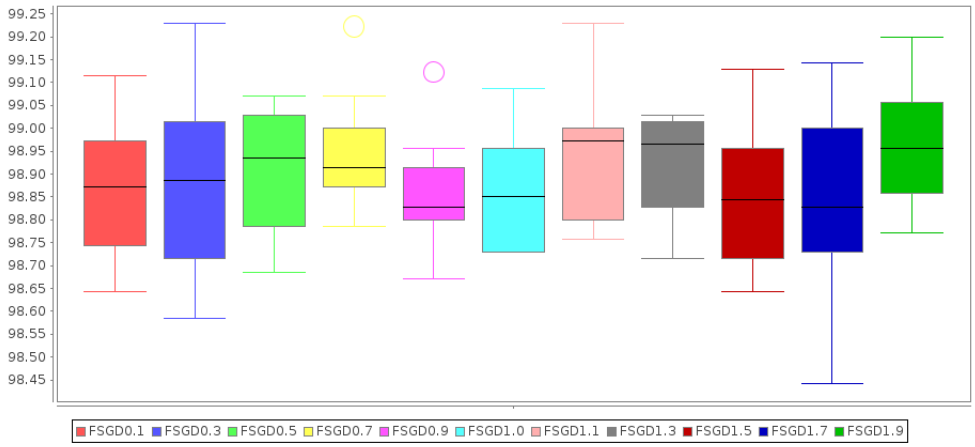

Figure 2. Experiment 1.2: Boxplots accuracies for FSGD with $\eta = \gamma = 0.1$, $\nu = 0.1, 0.3, \ldots, 1.9$ and 1.0.

From Experiment 1.2, it is deduced that the use of momentum contributes to better performance close to 99%, while FSGD without momentum in Experiment 1.1 barely reaches about 95.6% for the last fold.

3.1.3. Experiment 1.3

Other experiment considers FSGD with $\eta = 0.001$ and a high value for momentum $\gamma = 0.9$. The results are shown in Table 4 together with the boxplots of Figure 3. Additionally, in Table 5, the correlation of the columns of Table 4 are shown, and it is notorious the high correlation between all columns for different values of $\nu = 0.1$ to 1.9, which means that a high value of momentum and low learning rate diminishes the effect of the fractional factor f_w^ν. In fact, the correlation matrix of Table 5 makes this evident since all the correlation

values are higher than 0.91, in spite of the value of ν. Moreover, the performance for $\gamma = 0.9$ decreases about 2% with respect to Experiment 1.2 with lower momentum $\gamma = 0.1$, as is appreciated when comparing Tables 3 and 4.

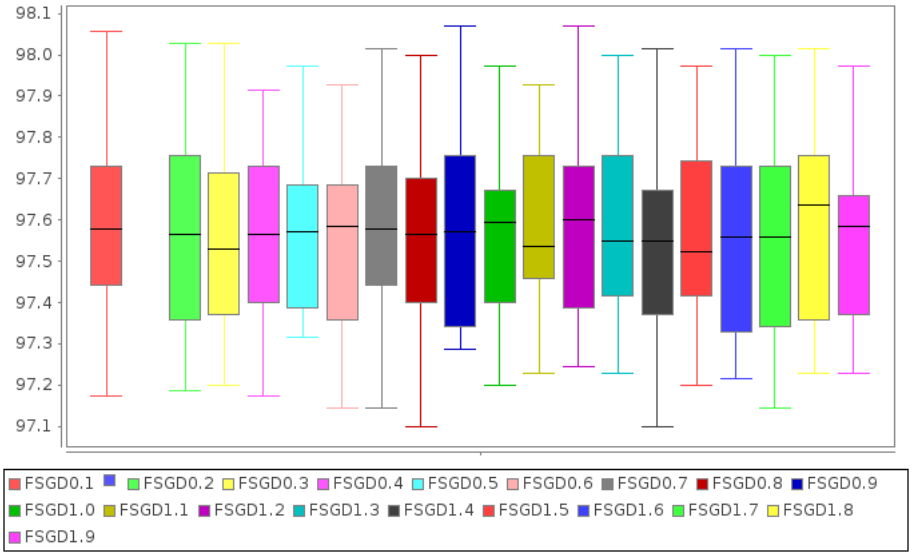

Figure 3. Accuracy boxplots for FSGD with 10-cross folding, momentum = 0.9 and $\nu = 0.1, 0.2, \ldots, 1.9$.

Table 4. Comparison between SGD and FSGD with momentum. The case $\nu = 1.0$ reduces to SGD.

FSGD with Momentum = 0.9 ($\nu = 0.1, \ldots, 1.9$)																			
0.1	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**1.7**	**1.8**	**1.9**
97.3	97.4	97.4	97.3	97.4	97.4	97.3	97.3	97.3	97.4	97.5	97.3	97.4	97.3	97.2	97.3	97.3	97.3	97.4
97.6	97.6	97.5	97.6	97.6	97.6	97.5	97.6	97.6	97.6	97.6	97.6	97.6	97.5	97.5	97.5	97.5	97.6	97.6
97.4	97.4	97.4	97.4	97.4	97.3	97.4	97.4	97.3	97.4	97.4	97.4	97.4	97.4	97.4	97.3	97.3	97.4	97.4
97.5	97.3	97.4	97.4	97.4	97.4	97.5	97.6	97.4	97.6	97.5	97.4	97.4	97.4	97.5	97.4	97.4	97.4	97.4
97.6	97.7	97.6	97.8	97.7	97.7	97.7	97.7	97.7	97.7	97.7	97.7	97.8	97.7	97.8	97.7	97.7	97.8	97.6
97.7	97.8	97.7	97.7	97.8	97.8	97.7	97.7	97.8	97.8	97.8	97.7	97.8	97.7	97.7	97.7	97.7	97.8	97.7
97.6	97.6	97.6	97.6	97.6	97.6	97.6	97.5	97.6	97.6	97.5	97.6	97.5	97.6	97.5	97.6	97.7	97.7	97.6
98.1	98.0	98.0	97.9	98.0	97.9	98.0	98.0	98.1	98.0	97.9	98.1	98.0	98.0	98.0	98.0	98.0	98.0	98.0
97.2	97.2	97.2	97.2	97.3	97.1	97.1	97.1	97.3	97.2	97.2	97.2	97.2	97.1	97.2	97.2	97.1	97.2	97.2
97.8	97.8	97.7	97.7	97.7	97.7	97.8	97.7	97.8	97.7	97.8	97.7	97.7	97.7	97.7	97.7	97.8	97.7	97.7

Table 5. Correlation matrix for FSGD: learning rate = 0.001, momentum = 0.9 and $\nu = 0.1, 0.2, \ldots, 1.9$.

									ν									
ν	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**1.7**	**1.8**	**1.9**
0.1	1																		
0.2	0.96	1																	
0.3	0.97	0.99	1																
0.4	0.93	0.96	0.95	1															
0.5	0.95	0.98	0.98	0.94	1														
0.6	0.95	0.98	0.97	0.97	0.97	1													
0.7	0.99	0.97	0.98	0.96	0.95	0.96	1												
0.8	0.98	0.94	0.95	0.96	0.92	0.94	0.98	1											
0.9	0.96	0.97	0.98	0.95	0.98	0.96	0.96	0.94	1										
1.0	0.97	0.94	0.95	0.95	0.94	0.96	0.97	0.98	0.95	1									
1.1	0.94	0.95	0.96	0.94	0.94	0.94	0.94	0.97	0.94	0.95	1								
1.2	0.96	0.98	0.98	0.96	0.98	0.96	0.97	0.94	0.98	0.94	0.93	1							
1.3	0.94	0.98	0.98	0.98	0.98	0.97	0.96	0.95	0.97	0.95	0.96	0.98	1						
1.4	0.98	0.97	0.98	0.97	0.95	0.97	0.99	0.98	0.97	0.98	0.96	0.97	0.97	1					
1.5	0.92	0.92	0.92	0.97	0.92	0.91	0.95	0.95	0.94	0.92	0.90	0.94	0.94	0.94	1				
1.6	0.96	0.97	0.98	0.97	0.97	0.96	0.97	0.95	0.98	0.94	0.92	0.98	0.97	0.98	0.96	1			
1.7	0.96	0.96	0.97	0.97	0.94	0.96	0.98	0.95	0.96	0.95	0.92	0.95	0.94	0.98	0.95	0.98	1		
1.8	0.95	0.97	0.97	0.96	0.98	0.99	0.96	0.93	0.97	0.95	0.91	0.97	0.97	0.96	0.94	0.98	0.97	1	
1.9	0.97	0.98	0.98	0.96	0.97	0.97	0.97	0.96	0.98	0.96	0.94	0.98	0.96	0.99	0.92	0.98	0.97	0.97	1

3.2. Experiment 2

Experiment 2 uses the HAR dataset Actitracker [29]. It was released by Wireless Sensor Data Mining (WISDM) lab and refers to 36 users using a smartphone in their pocket at a sample rate of 20 samples per second. The dataset contains acceleration values for x, y and z axes, while the user performs six different activities in a controlled environment: downstairs, jogging, sitting, standing, upstairs, and walking. The number of samples is 1,098,209, which was originally split into 80% for training and 20% for testing.

To obtain better experimental support, these data were merged, and cross-validation with $K = 4$ folds was applied with shuffle.

The source code was adapted from [31], and it considers a 2D-convolutional neural network (2D-CNN) with two dense layers and ReLu activation function, followed by a softmax layer.

Fractional optimizers were studied in two groups based on their performance; the first group is FSGD, FSGDP, FAdagrad and FAdadelta, and the second group is FRMSProp, FAdam and FAdamP. It gives place to Experiments 2.1 and 2.2.

3.2.1. Experiment 2.1

The source code of [31] was modified to include FSGD, FSGDP, FAdagrad and FAdadelta.

Because of space saving, the boxplots of accuracies for $K = 4$ folds are illustrated in Figures 4–7, but tables with numerical data are not included. The following observations can be made:

- Figure 4. The highest score for FSGD is 82.8% at $\nu = 1.7$.
- Figure 5. The highest score for FSGDP is 82.71% at $\nu = 1.7$ (a marginal difference with respect to FSGD).
- Figure 6. FAdagrad just reaches its maximum 84.64% at $\nu = 1.6$.
- Figure 7. The worst performance of these four optimizers is for FAdadelta with a maximum of 63.11% at $\nu = 1.8$.

In this experiment, is obvious the influence of the fractional factor f_w^ν to enhance the performance compared with the traditional first-order case. However, these results are not the best possible because they can be improved by other optimizers, as will be shown in the next experiment.

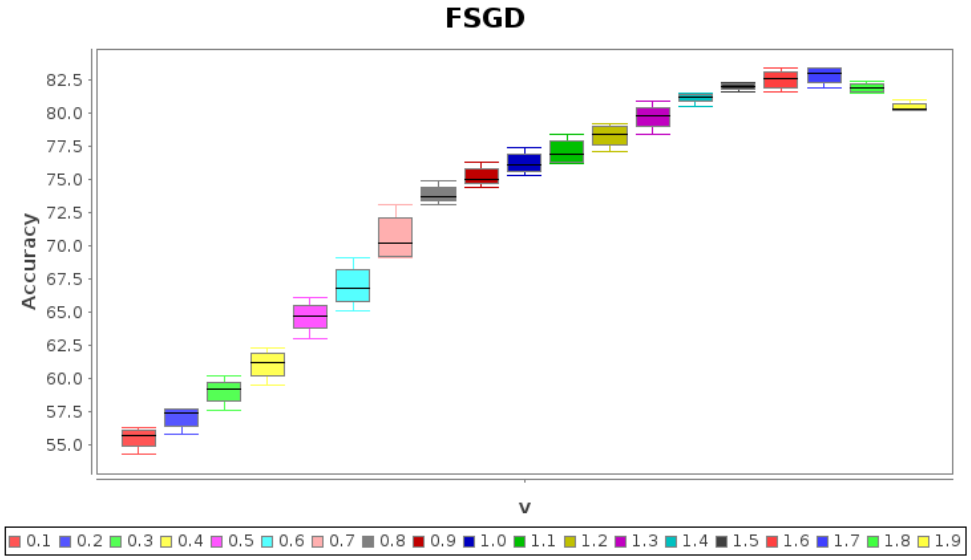

Figure 4. Experiment 2.1: FSGD cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

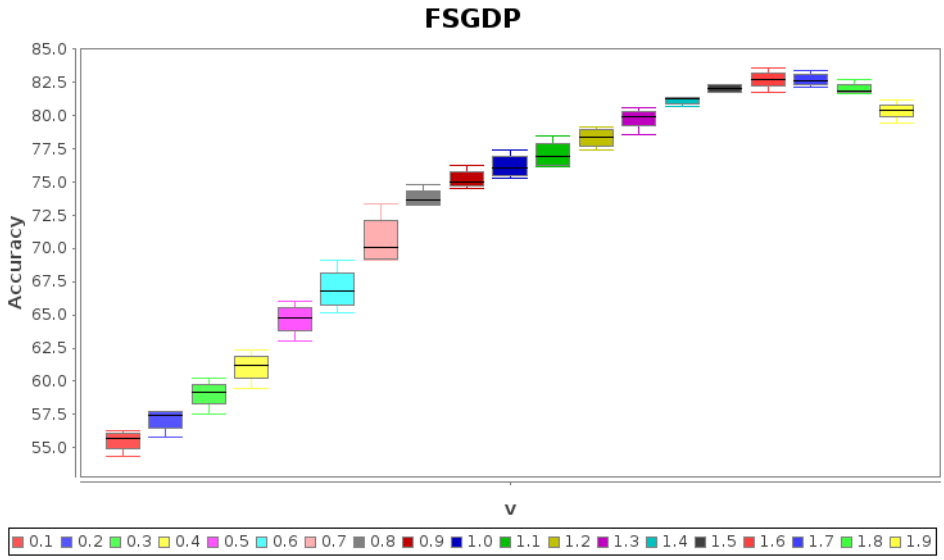

Figure 5. Experiment 2.1: FSGDP cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

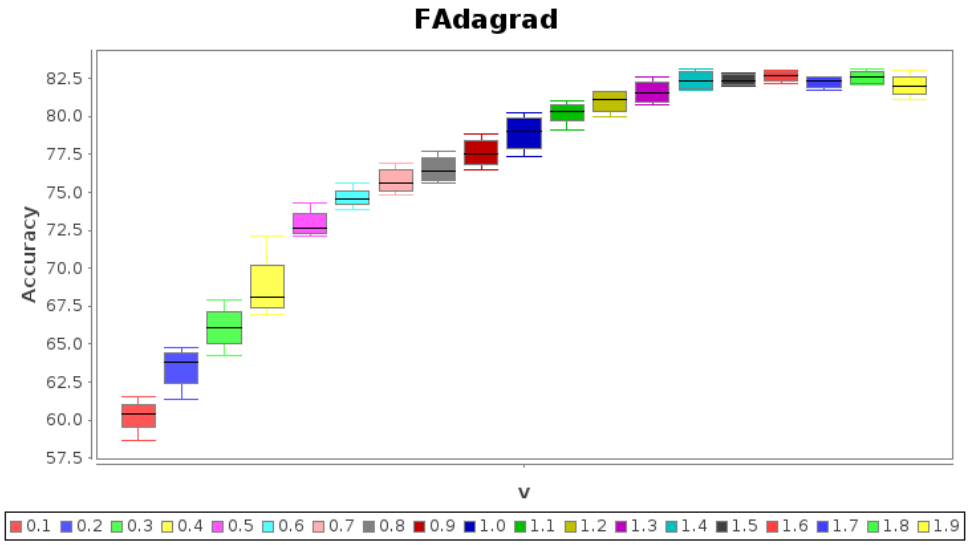

Figure 6. Experiment 2.1: FAdagrad cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

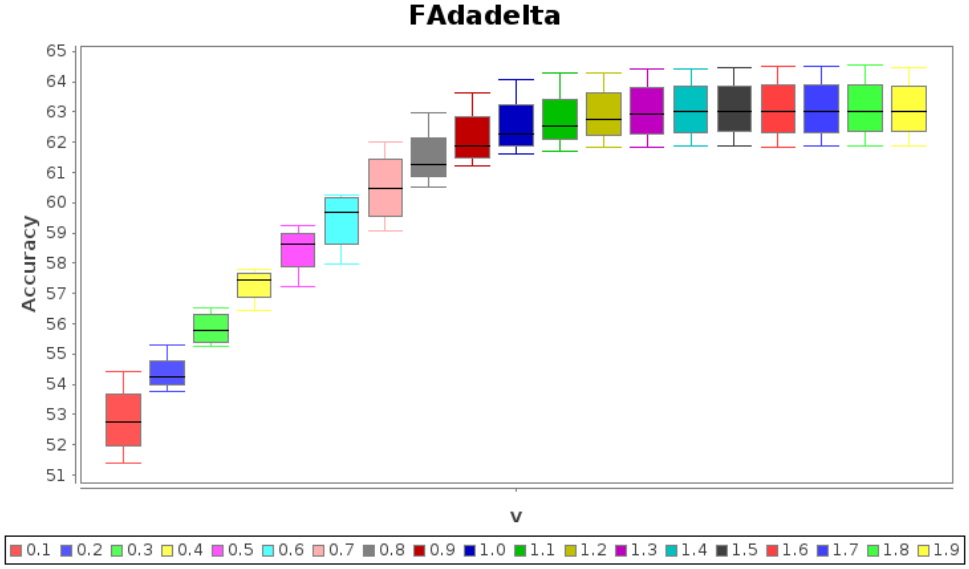

Figure 7. Experiment 2.1: FAdadelta crossfolding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

3.2.2. Experiment 2.2

In Experiment 2.2, the modification to the source code of [31] was to include fractional FRMSProp, FAdam and FadamP versions.

Figures 8–10 show the boxplots from accuracies of the $K = 4$ folds for FRMSProp, FAdam and FadamP, respectively, with ν in the interval 0.1 to 1.9 and step equals to 0.1.

In Figure 8, a few candidates can be identified as "the winners", although there is no a strong dominant case. The most relevant cases of Figure 8 are

- FRSMProp at $\nu = 0.5$. This case has a superior and more compact behavior compared with the integer case $v = 1.0$.
- FRSMProp at $\nu = 1.4$. This case has similar performance to that of RMSProp ($\nu = 1.0$, average 89% and standard deviation 1.68), but the case $\nu = 1.4$ is slightly higher (average 89.5% and standard deviation 1.52).

In Figure 9, FAdam with $\nu = 1.1$ can be considered better than for $\nu = 1.0$ since the first has an average accuracy equals to 87.7 (1.15% over the case $\nu = 1.0$) although it certainly has a slightly larger standard deviation (+0.43) than for the case $\nu = 1.0$.

According to Figure 10, FAdamP with $\nu = 1$ (i.e., AdamP) can be considered the best (just in FadamP category) since is not possible to identify a convincingly better case than accuracy 88% and standard deviation = 0.84. However, when comparing group 2, FadamP is improved by FRSMProp at $\nu = 1.4$.

In general, for this Experiment 2.2, the best cases correspond to FRSMProp, but it is also important to mention that group 2 outperformed the group 1 accuracies of Experiment 2.1.

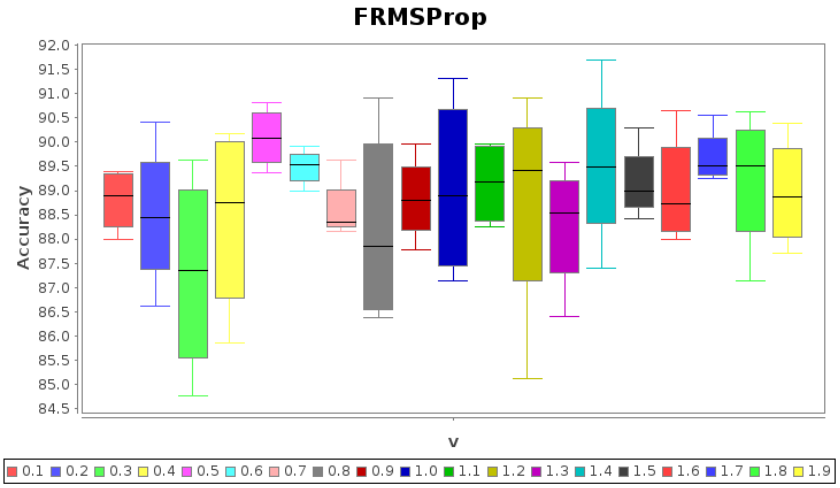

Figure 8. Experiment 2.2: FRMSProp cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

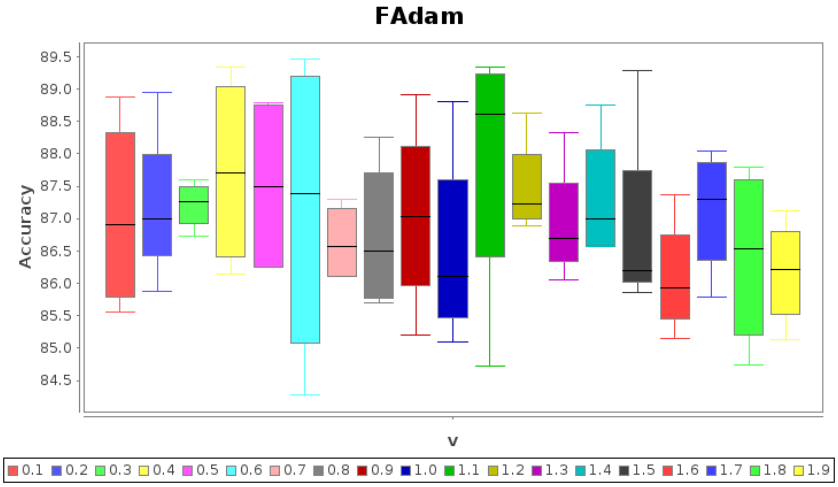

Figure 9. Experiment 2.2: FAdam cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

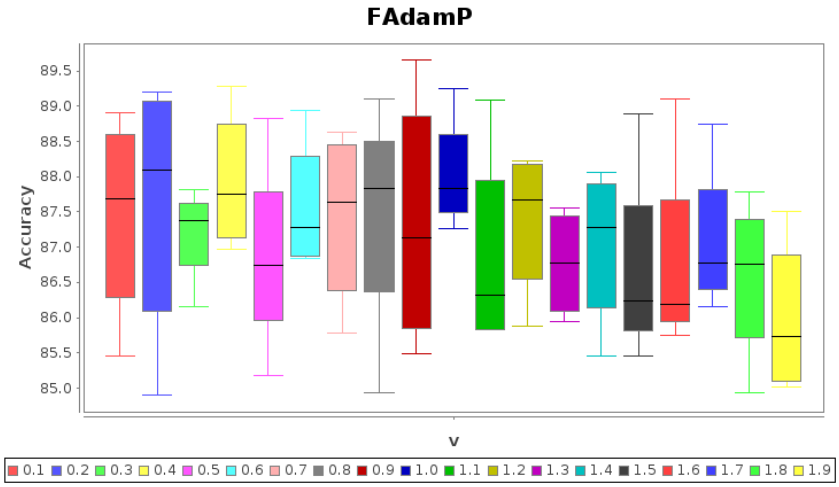

Figure 10. Experiment 2.2: FAdamP cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

3.3. Experiment 3

In Experiment 3, the dataset "Human Activity Recognition Using Smartphones" [30] was used. It contains data from 30 subjects performing one of six activities: walking, walking upstairs, walking downstairs, sitting, standing and laying. The data were collected while wearing a waist-mounted smartphone, and the movement labels were manually obtained from videos.

Originally, the dataset was split into 70% for training and 30% for testing. Both training and testing subsets were mixed to obtain a unified dataset and to apply cross-validation with $K = 4$ folds and shuffle, that produces a fold size of 25% of the whole, which yields to 75% for training and 25% for testing, trying to match the original split of 70% + 30%.

The optimizers were studied with the same groups of Experiment 2, because of their behavior.

3.3.1. Experiment 3.1

The source code was adapted from [32] to include optimizers FSGD, FSGDP, FAdagrad and FAdadelta. Crossfolding was applied for each optimizer, moving ν from 0.1 to 1.9 with steps of 0.1.

The learning rate was 0.001 and the momentum γ was equal to 0.1. The results for FSGD, FSGDP, FAdagrad and FAdadelta are shown in Figures 11–14 as boxplots, where it is possible to see essentially the same tendency for each optimizer: increasing conform ν increases to reach a maximum and then decreases. FSGD and FSGDP decrease abruptly for $\nu \geq 1.7$ (see Figures 11 and 12). The highest average accuracies are at $\nu = 1.6$ for FSGD and FSGDP with accuracies of 76.3% and 76.4%, respectively.

In the case of Figure 13 for FAdagrad, the maximum average accuracy is 80.9% at $\nu = 1.7$. FAdadelta in Figure 14 presents the worst performance, given that the maximum is 53.5% at $\nu = 1.8$.

Again, in this experiment, it is possible to appreciate the influence of the fractional factor f_w^ν to modify the accuracy.

3.3.2. Experiment 3.2

In Figures 15–17, boxplots are shown for the accuracies of $K = 4$ folds of FRMSProp, FAdam and FadamP respectively.

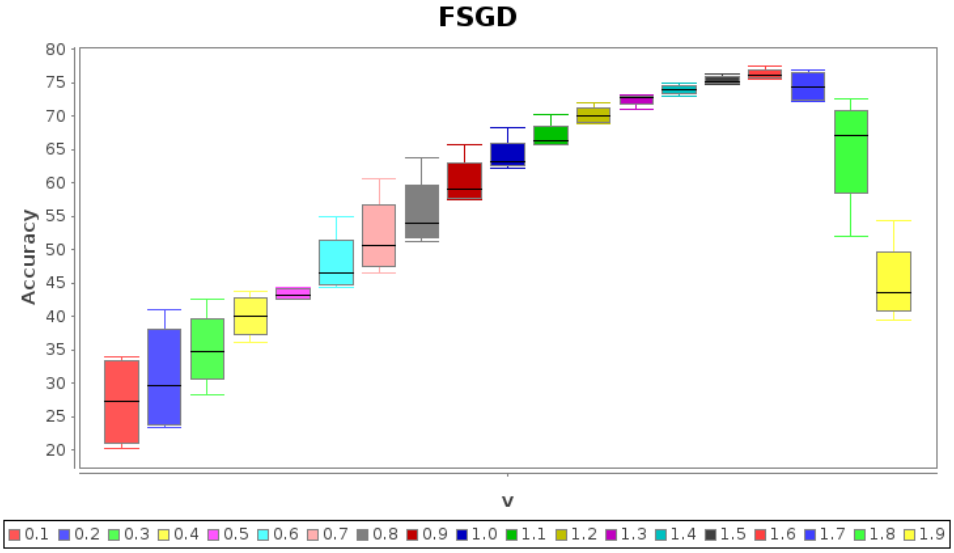

Figure 11. Experiment 3.1: FSGD cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

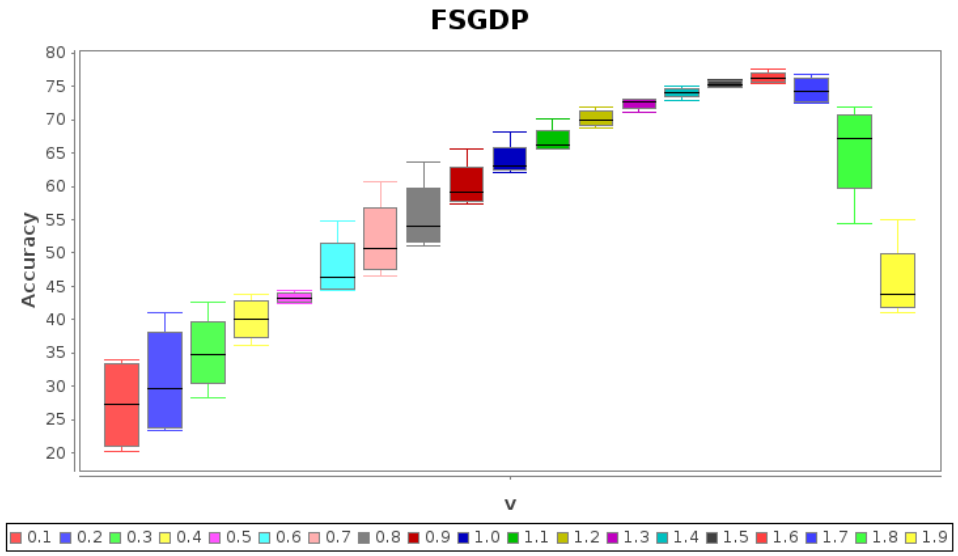

Figure 12. Experiment 3.1: FSGDP cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

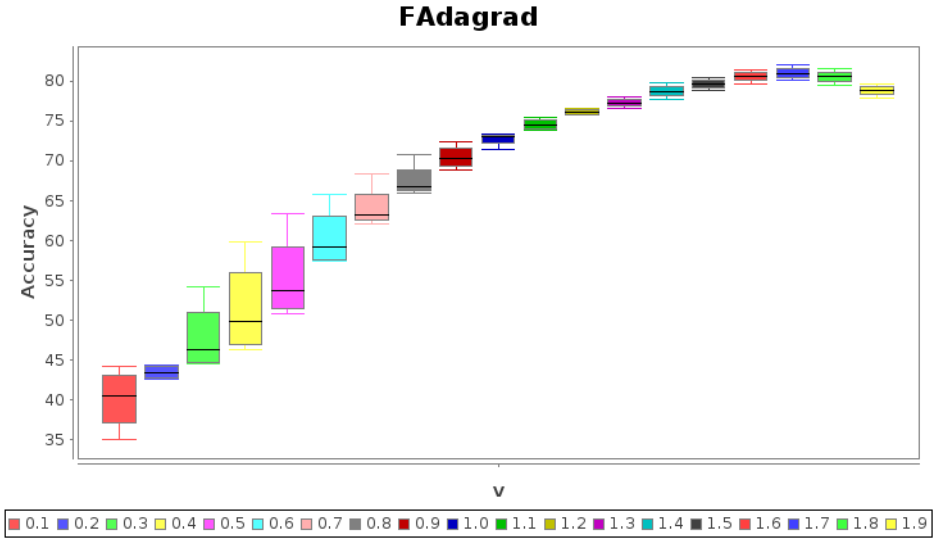

Figure 13. Experiment 3.1: FAdagrad cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

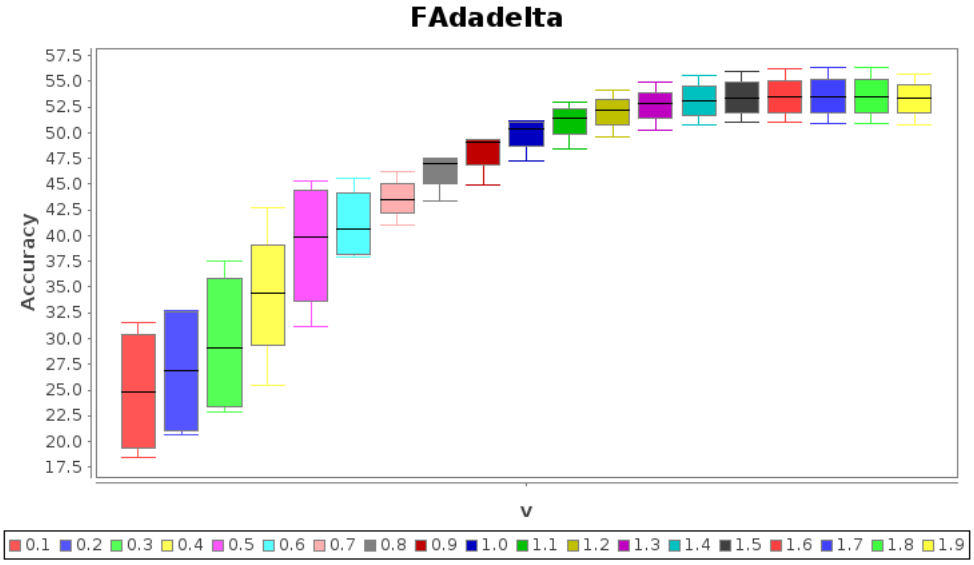

Figure 14. Experiment 3.1: FAdadelta cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

From these three figures, the next relevant cases were observed for each case independently:

- FRMSProp. In Figure 15, the case $\nu = 1.0$ is improved by the rest of the cases, except $\nu = 1.8$.
- Fadam. In Figure 16, the case $\nu = 1.0$ is improved by the accuracy at $\nu = 1.2$.
- FadamP. In Figure 17, the case $\nu = 1.0$ is improved by the accuracy at $\nu = 0.9$.

In this experiment, FRMSProp does not have better performance than FAdam and FAdamP; however, the fractional order seems to slightly modify the performance, and in most cases, a value other than 1.0 provides better accuracy.

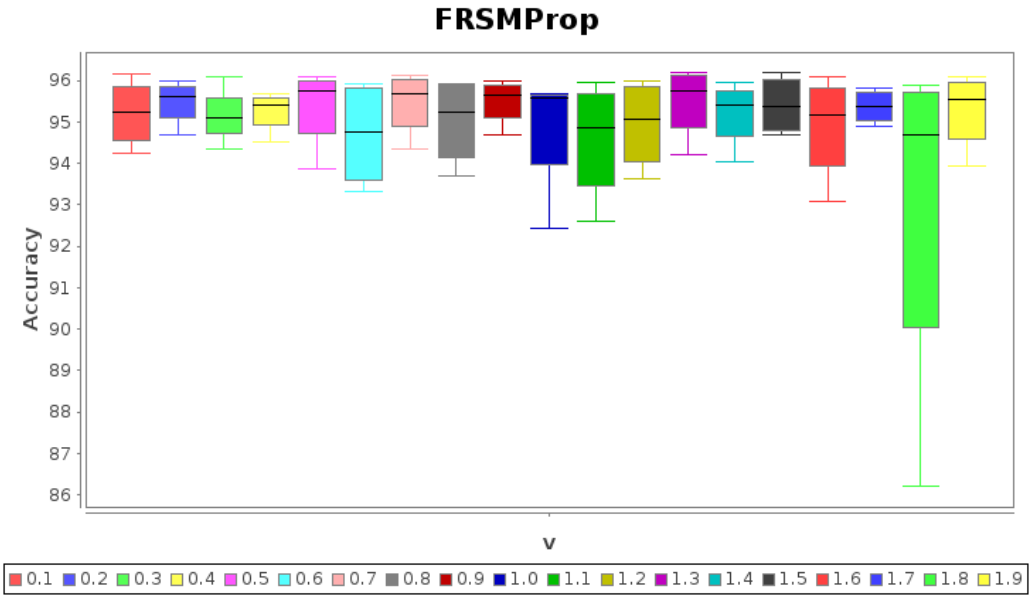

Figure 15. Experiment 3.2: FRMSProp cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

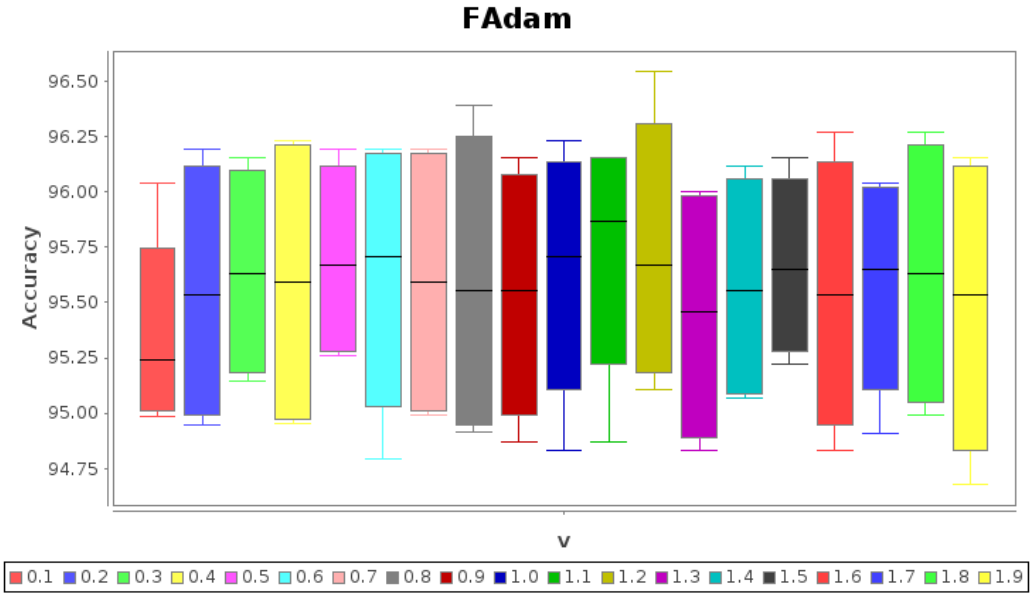

Figure 16. Experiment 3.2: FAdam cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

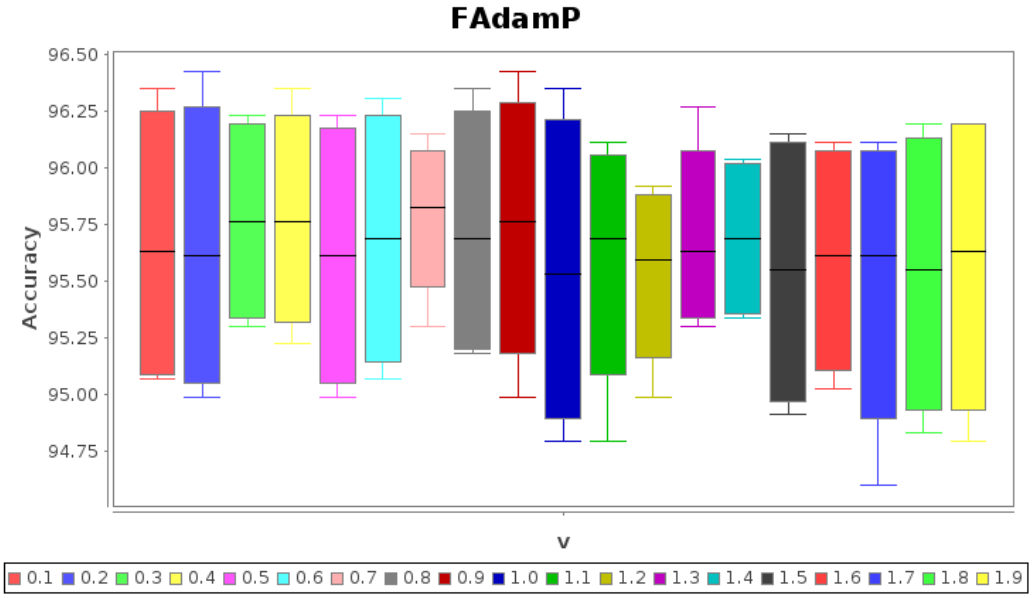

Figure 17. Experiment 3.2: FAdamP cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

3.3.3. Experiment 3.3

Finally, the same experiments 3.2 and 3.2 were repeated with 10 folds, and similar results were obtained. An overview of the accuracies of both groups, group 1 (FSGD, FSGDP, FAdagrad, FAdadelta) and group 2 (FRMSProp, Fadam, FAdamP), is shown as boxplots in Figures 18 and 19. The boxplots correspond to each optimizer listed in groups 1 and 2, for $\nu \in [0.1, 1.9]$ with increments of 0.1.

In Figure 18, it is possible to appreciate the highest accuracy for FAdagrad at $\nu = 1.7$, whereas in Figure 19, the highest accuracy is for FAdam at $\nu = 1.2$.

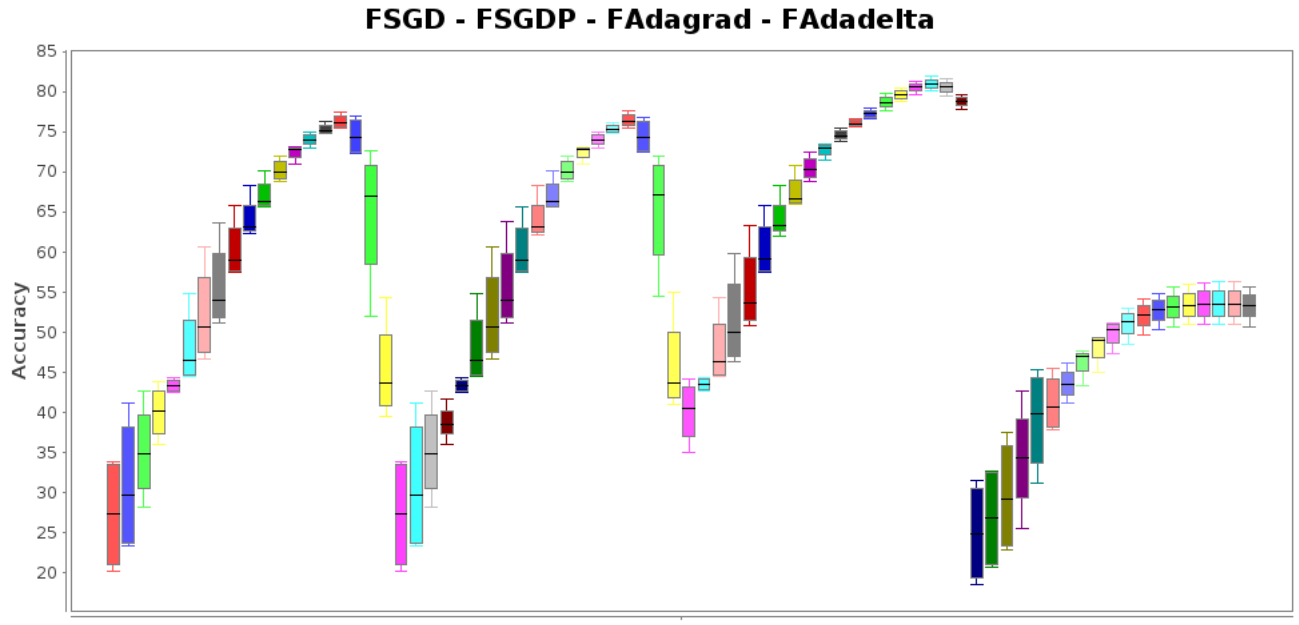

Figure 18. Experiment 3.2: Group 1, cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

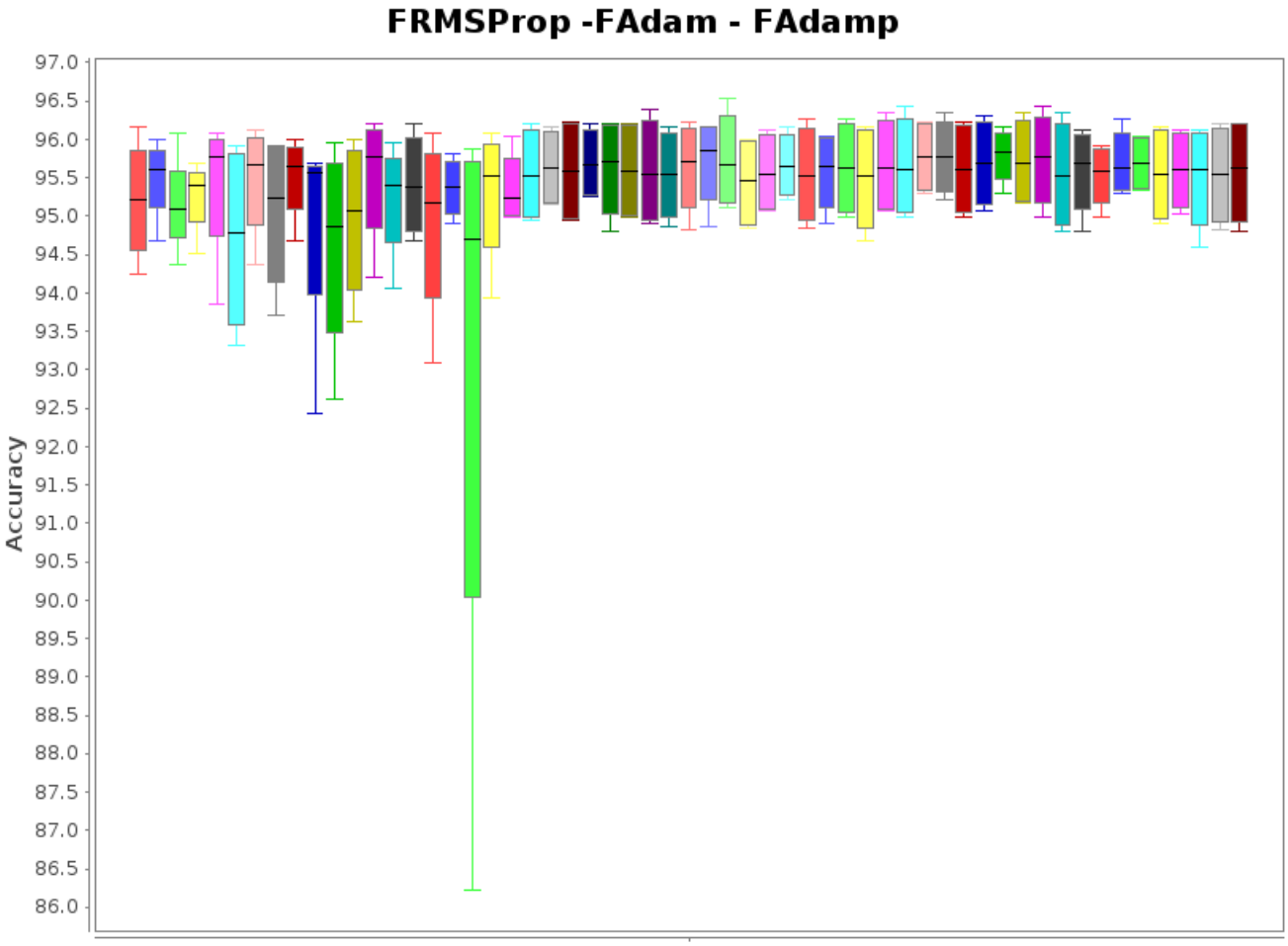

Figure 19. Experiment 3.2: Group 2, cross-folding accuracies, $K = 4$ folds, $\nu \in [0.1, 1.9]$.

Once the experiments have been carried out, it is convenient to mention that, similar to the integer case, a geometric interpretation can be given to Equation (24) as a gradient

at the region of interest on the error surface. The main benefit of fractional v-derivative over the integer version is the expansion of the region of search around the test point given by v and the weights w_{kj}^{l} [33]. In this sense, the benefit is provided by the factor f_w^v of Equation (25), and to explore the effect of non-locality, a 3D plot is shown in Figure 20. It is possible to identify four regions with different behavior:

1. Region A: For $v \to 0^+$. The factor f_w^v follows $|w_{kj}^{l}|$ and it could produce divergence for w_{kj}^{l} and consequently for f_w^v, and for the fractional gradient $D^{v}_{w_{kj}^{l}} E_i$.
2. Region B: For $v = 1$. The integer case corresponds to the red line $f_w^v = 1$. No context information is considered, just the local point.
3. Region C: For $1 < v < 2$. The surface f_w^v reaches a local maximum $f_w^{v^*} = 8.75$ at $v^* = 1.77$ for $\epsilon = 0.01$.
4. Region D: For $v \to 2^-$. f_w^v tends to zero as fast as $|w_{kj}^{l}|$ increases. It promotes that small weights increase its values and move to the flatter region, where they will stabilize.

The Cartesian axes in Figure 20 are as follows:

- x-axis. The weights w_{kj}^{l}.
- y-axis. Fractional value $v \in (0, 2)$.
- z-axis. The fractional factor f_w^v that modifies the integer order gradient.

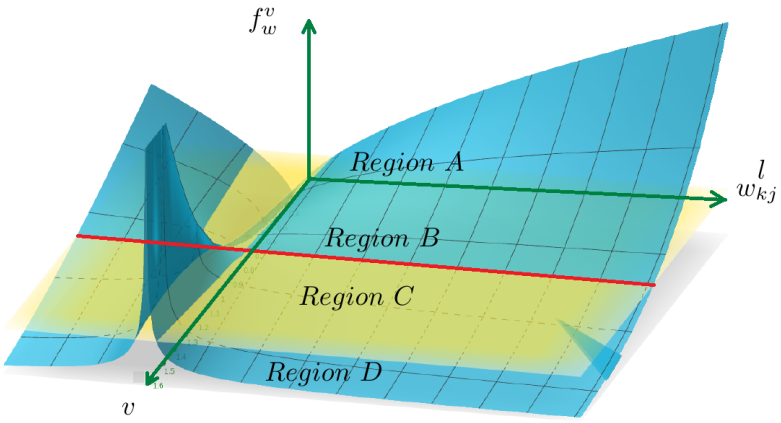

Figure 20. Plot for the fractional factor f_w^v with $\epsilon = 0.01$

In Figure 20 there is a yellow plane at $f_w^v = 1$ used as reference for the red line of the integer case.

It is noteworthy that in region C, the experimental accuracies of the fractional optimizers (depending of v) follow a similar behavior to f_w^v. For now, it is just an observation that merits exploring the possible relationship between the optimal fractional order v^* and the region of best accuracy for fractional optimizers.

4. Discussion

Unlike other optimizers that have been proposed, alluding mainly to concepts such as momentum or velocity, in this paper, gradient descent variants are proposed based on fractional calculus concepts, and specifically on the Caputo fractional derivative.

The proposed fractional optimizers add the prefix "F" to original names, and their update formulas essentially aggregate the f_w^v factor defined in such a way that the limit exists when the v-order derivative tends to 1, which leads to a more general formula that includes the integer order as a special case.

Fractional optimizers are slightly more expensive computationally because they require computing the f_w^v factor. However, they are still very competitive because the computation is performed efficiently through Tensorflow, and the additional advantage is that the fractional factor transfers non-local exploring properties, rather than just considering an infinitely small neighborhood around a point on the error surface as in the

integer case, which experimentally is traduced in the improvement of the performance of fractional optimizers.

The fractional factor f_w^v provides an additional degree of freedom to the backpropagation algorithm, and consequently, to the learning capacity of neural networks, as was shown in several experiments.

The fractional optimizers were successfully implemented in Tensorflow–Keras with modifications to the original source code to obtain FSGD, FSGDP, FAdagrad, FAdadelta, FRMSProp, FAdam and FAdamP classes. Everything indicates that it is possible to apply the same methodology to modify other gradient-based optimizers, as well as making implementations in other frameworks.

Three experiments were carried out with MNIST and two HAR datasets. The results on crossfolding show that in all the experiments, a fractional order provides better performance than the first order for the same neural network architectures.

In the experiments, FSGD, FSGDP, FAdagrad and FAdadelta (group 1) basically follow the same pattern of increasing their performance as the v-order does, obtaining a maximum and then decreasing.

Other optimizers, such as FRMSProp, FAdam and FAdamP (group 2), do not follow the same pattern, and seem to be less susceptible to the fractional order change. From the experiments, it can be said that FRMSProp has an "intermediate" pattern between the optimizers of group 1 and group 2.

Even so, essentially in all the experiments, the best performing derivative order was a fractional value.

Therefore, based on the results, it is possible to affirm that fractional derivative gradient optimizers can help to improve the performance on the training and validation task, and opens the possibility to include more fractional calculus concepts to neural networks applied to HAR.

The Tensorflow–Keras implementations of this work are available in a repository to contribute to the deep learning and HAR communities to improve and apply these techniques based on fractional calculus.

Future works include exploring more fractional derivative definitions and HAR datasets with fractional regularization factors as well as studying the effects in vanishing gradient problems with other neural network architectures.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The Tensorflow-Keras implementations of this work are available at http://ia.azc.uam.mx/.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Haykin, S.S. *Neural Networks and Learning Machines*, 3rd ed.; Pearson Education: Upper Saddle River, NJ, USA, 2009.
2. Ruder, S. An overview of gradient descent optimization algorithms. *arXiv* **2016**, arXiv:1609.04747.
3. Duchi, J.; Hazan, E.; Singer, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.* **2011**, *12*, 2121–2159.
4. Zeiler, M.D. ADADELTA: An Adaptive Learning Rate Method. *arXiv* **2012**, arXiv:1212.5701.
5. Tieleman, T.; Hinton, G. *Neural Networks for Machine Learning*; Technical Report; COURSERA: Mountain View, CA, USA, 2012.
6. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization, 2014. In Proceedings of the 3rd International Conference for Learning Representations, San Diego, CA, USA, 7–9 May 2015.
7. Heo, B.; Chun, S.; Oh, S.J.; Han, D.; Yun, S.; Kim, G.; Uh, Y.; Ha, J.W. AdamP: Slowing Down the Slowdown for Momentum Optimizers on Scale-invariant Weights. In Proceedings of the International Conference on Learning Representations (ICLR), Virtual Event, 3–7 May 2021.
8. Podlubny, I. Mathematics in Science and Engineering. In *Fractional Differential Equations*; Academic Press: Cambridge, MA, USA, 1999; Volume 198, p. 340.

9. Oustaloup, A. *La dérivation non Entière: Théorie, Synthèse et Applications*; Hermes Science Publications: New Castle, PA, USA, 1995; p. 508.
10. Luchko, Y. *Fractional Integrals and Derivatives: "True" versus "False"*; MDPI: Basel, Switzerland, 2021. [CrossRef]
11. Miller, K.; Ross, B. *An Introduction to the Fractional Calculus and Fractional Differential Equations*; Wiley: Hoboken, NJ, USA, 1993.
12. Bao, C.; Pu, Y.; Zhang, Y. Fractional-Order Deep Backpropagation Neural Network. *Comput. Intell. Neurosci.* **2018**, *2018*, 7361628. [CrossRef] [PubMed]
13. Wang, J.; Wen, Y.; Gou, Y.; Ye, Z.; Chen, H. Fractional-order gradient descent learning of BP neural networks with Caputo derivative. *Neural Netw.* **2017**, *89*, 19–30. [CrossRef] [PubMed]
14. Machado, J.T.; Kiryakova, V.; Mainardi, F. Recent history of fractional calculus. *Commun. Nonlinear Sci. Numer. Simul.* **2011**, *16*, 1140–1153. [CrossRef]
15. Mainardi, F. *Fractional Calculus and Waves in Linear Viscoelasticity*, 2nd ed.; World Scientific: Singapore, 2022; Number 2, p. 628.
16. Muresan, C.I.; Birs, I.; Ionescu, C.; Dulf, E.H.; De Keyser, R. A Review of Recent Developments in Autotuning Methods for Fractional-Order Controllers. *Fractal Fract.* **2022**, *6*, 37. [CrossRef]
17. Yousefi, F.; Rivaz, A.; Chen, W. The construction of operational matrix of fractional integration for solving fractional differential and integro-differential equations. *Neural Comput. Appl.* **2019**, *31*, 1867–1878. [CrossRef]
18. Gonzalez, E.A.; Petráš, I. Advances in fractional calculus: Control and signal processing applications. In Proceedings of the 2015 16th International Carpathian Control Conference (ICCC), Szilvasvarad, Hungary, 27–30 May 2015; pp. 147–152. [CrossRef]
19. Henriques, M.; Valério, D.; Gordo, P.; Melicio, R. Fractional-Order Colour Image Processing. *Mathematics* **2021**, *9*, 457. [CrossRef]
20. Shoaib, B.; Qureshi, I.M.; Shafqatullah; Ihsanulhaq. Adaptive step-size modified fractional least mean square algorithm for chaotic time series prediction. *Chin. Phys. B* **2014**, *23*, 050503. [CrossRef]
21. Tarasov, V.E. On History of Mathematical Economics: Application of Fractional Calculus. *Mathematics* **2019**, *7*, 509. [CrossRef]
22. Alzabut, J.; Tyagi, S.; Abbas, S. Discrete Fractional-Order BAM Neural Networks with Leakage Delay: Existence and Stability Results. *Asian J. Control* **2020**, *22*, 143–155. [CrossRef]
23. Ames, W.F. Chapter 2—Fractional Derivatives and Integrals. In *Fractional Differential Equations*; Podlubny, I., Ed.; Mathematics in Science and Engineering; Elsevier: Amsterdam, The Netherlands, 1999; Volume 198, pp. 41–119. [CrossRef]
24. Garrappa, R.; Kaslik, E.; Popolizio, M. Evaluation of Fractional Integrals and Derivatives of Elementary Functions: Overview and Tutorial. *Mathematics* **2019**, *7*, 407. [CrossRef]
25. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: tensorflow.org (accessed on 8 September 2022).
26. Chollet, F.; Zhu, Q.S.; Rahman, F.; Lee, T.; Marmiesse, G.; Zabluda, O.; Qian, C.; Jin, H.; Watson, M.; Chao, R.; et al. Keras. 2015. Available online: https://keras.io/ (accessed on 4 July 2022).
27. Google Inc. Google Colab. 2015. Available online: https://colab.research.google.com (accessed on 4 July 2022).
28. Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Process. Mag.* **2012**, *29*, 141–142. [CrossRef]
29. Actitracker. Available online: http://www.cis.fordham.edu/wisdm/dataset.php (accessed on 4 July 2022).
30. Reyes-Ortiz, J.L.; Anguita, D.; Ghio, A.; Oneto, L.; Parra, X. Human Activity Recognition Using Smartphones Dataset. Available online: https://archive.ics.uci.edu/ml/machine-learning-databases/00240 (accessed on 4 July 2022).
31. HAR Using CNN in Keras. Available online: https://github.com/Shahnawax/HAR-CNN-Keras (accessed on 4 July 2022).
32. Jason, B. How to Model Human Activity from Smartphone Data. Available online: https://machinelearningmastery.com/how-to-model-human-activity-from-smartphone-data/ (accessed on 4 July 2022).
33. Khan, S.; Ahmad, J.; Naseem, I.; Moinuddin, M. A Novel Fractional Gradient-Based Learning Algorithm for Recurrent Neural Networks. *Circuits Syst. Signal Process.* **2018**, *37*, 593–612. [CrossRef]