**Gravity Optimizer**

**Dariush Bahrami 1; Sadegh Pouriyan Zadeh 2**

1 Faculty of New Sciences & Technologies, University of Tehran, Tehran, Iran

Email: dariush.bahrami@ut.ac.ir

2 Faculty of New Sciences & Technologies, University of Tehran, Tehran, Iran

Email: spouriyanz@ut.ac.ir

# Abstract

**# TODO: نوشتن خلاصه و کلمات کلیدی تو مرحله آخر**

Keywords:

# 1. Introduction

The question of choosing an adequate optimizer for a deep learning problem is not answered yet. Instead, there are ways like empirical comparing [1–3] or benchmarking [4] which help to find better configurations for optimization.

The most common optimization techniques in deep learning are SGD (Stochastic Gradient Descent)[5], RMS Prop[6], and Adam[7]. Table 1 shows the most common standard optimization techniques or optimizers in chronicle order.

Table 1. Common standard optimizers in deep learning in chronicle order

|  |  |
| --- | --- |
| Year Published | Optimization technique |
| 1951 [5] | SGD |
| 1964 [8] | SGD with momentum |
| 2011 [9] | AdaGrad |
| 2012 [10] | AdaDelta |
| 2012 [6] | RMSProp |
| 2013 [11] | SGD with Nesterov momentum |
| 2015 [7] | Adam |
| 2015 [7] | AdaMax |
| 2016 [12] | Nadam |
| 2018 [13] | AMSGrad |

# TODO: اینو تا 2 پاراگراف دیگه باید ادامه بدی

The rest of this article consists of the following sections. [Section 2](#_2_Gravity_Optimizer) describes the theory and mathematics of Gravity optimizer. In the following, the behavior and effect of each hyper-parameter are explained and at the end of this section, the best hyper-parameters obtained are suggested. [Section 3](#_3._Benchmark_Configuration) presents the tools used for the benchmark (including hardware, framework, and data set) and the architecture chosen. Then the settings used in the optimizers (including hyper-parameters) are reported in detail. In [Section 4](#_4._Results), the results obtained from the training of each dataset on the selected architecture are reported in its subsection. At the end of each subsection, the performance of the gravity optimizer is analyzed relative to the other two standard optimizers used. The final section provides a conclusion and what needs to be done in the future on Gravity optimizer.

# 2 Gravity Optimizer Design

This section provides ideas, theories, and mathematics about Gravity optimizer. By considering an inclined plane and using basic kinematic physics, an interesting analogy can be found between the gradients and the slope angle. In this analogy, the loss reduction is equivalent to the height of the rolling ball. [Fig. 1](#Figure_1) shows the schematic of the idea.

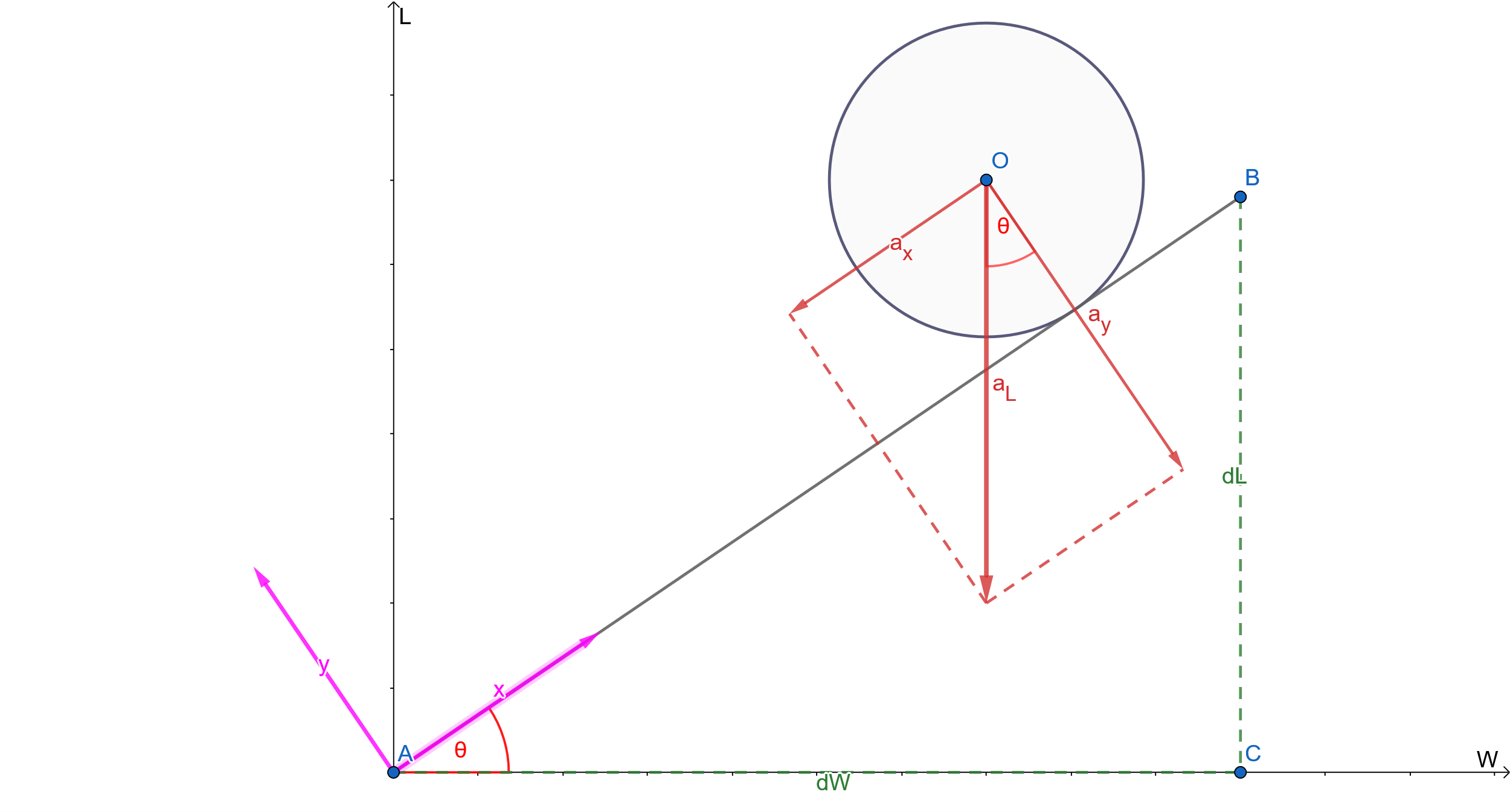


Figure 1. The global coordinate system and the inclined plane’s local coordinate system

For simplicity, the ball is treated as a point mass. The slope angle of the inclined plane, , is obtained by tangent definition:

(1)

For the sake of brevity is defined as follow:

(2)

A universal acceleration in the L axis direction, , is defined to imitate gravity which the idea of the name of Gravity optimizer came. The relationship between the inclined plane's local coordinate system,, and the global coordinate system, , can be obtained by using basic trigonometric relationships:

(2)

(3)

Then the relationship between acceleration in the mentioned coordinate systems can be written as:

(4)

Two parameters on the right side of [Eq. 4](#Equation_4), and , does not change with time. Therefore position equation at constant acceleration [] can be written as:

(5)

Assuming and simplifying gives:

(6)

Then by substituting [Eq. 6](#Equation_6) into [Eq. 2](#Equation_2):

(7)

which, together with [Eq. 4](#Equation_4) gives:

(8)

From [] we have trigonometric equations of sine and cosine of :

(9)

(10)

Thus using [Eq. 1](#Equation_1), [Eq. 8](#Equation_8), [Eq. 9](#Equation_9), and [Eq. 10](#Equation_10), finally gives as:

(11)

[Eq. 11](#Equation_11) is the parameter-update equation. As can be seen, there are a lot of hyper-parameters in this equation that needs a lot of time to be tuned. Besides, they are not intuitive. In the following subsections, these hyper-parameters will be replaced with more familiar and common hyper-parameters.

## 2.1 Learning Rate

In deep learning models, it is common to work with more familiar hyper-parameters like learning rate. To have more common hyper-parameters, the learning rate is defined in [Eq. 11](#Equation_11) as follows:

(12)

Substituting [Eq. 12](#Equation_12) into [Eq. 11](#Equation_11) gives:

(13)

To learn more about the above equation, look at back-prop optimization-based methods as a function of the gradient. Each optimization method takes the gradient as input and gives a step in the output for parameter changes (weight, bias, etc.). For example, Vanilla Gradient Descent is a linear function of the gradient:

(14)

which is plotted as follows:

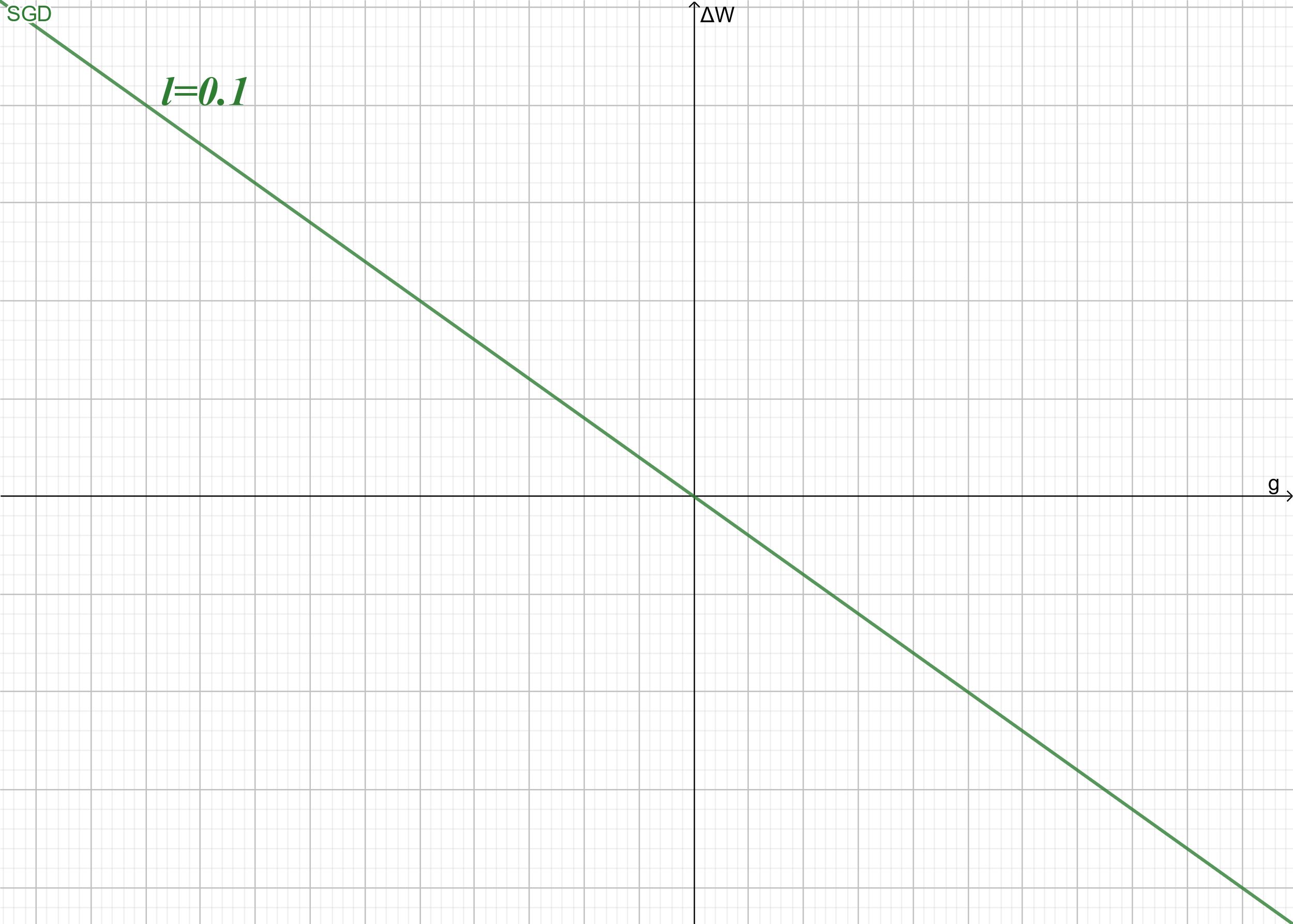


Figure 2. Vanilla Gradient Descent as a linear function of the gradient

To compare Gravity and GD optimizers, Gravity optimization function in [Eq. 13](#Equation_13) and Vanilla Gradient Descent is plotted in [Fig. 3](#Figure_3) for a given value of the learning rate:

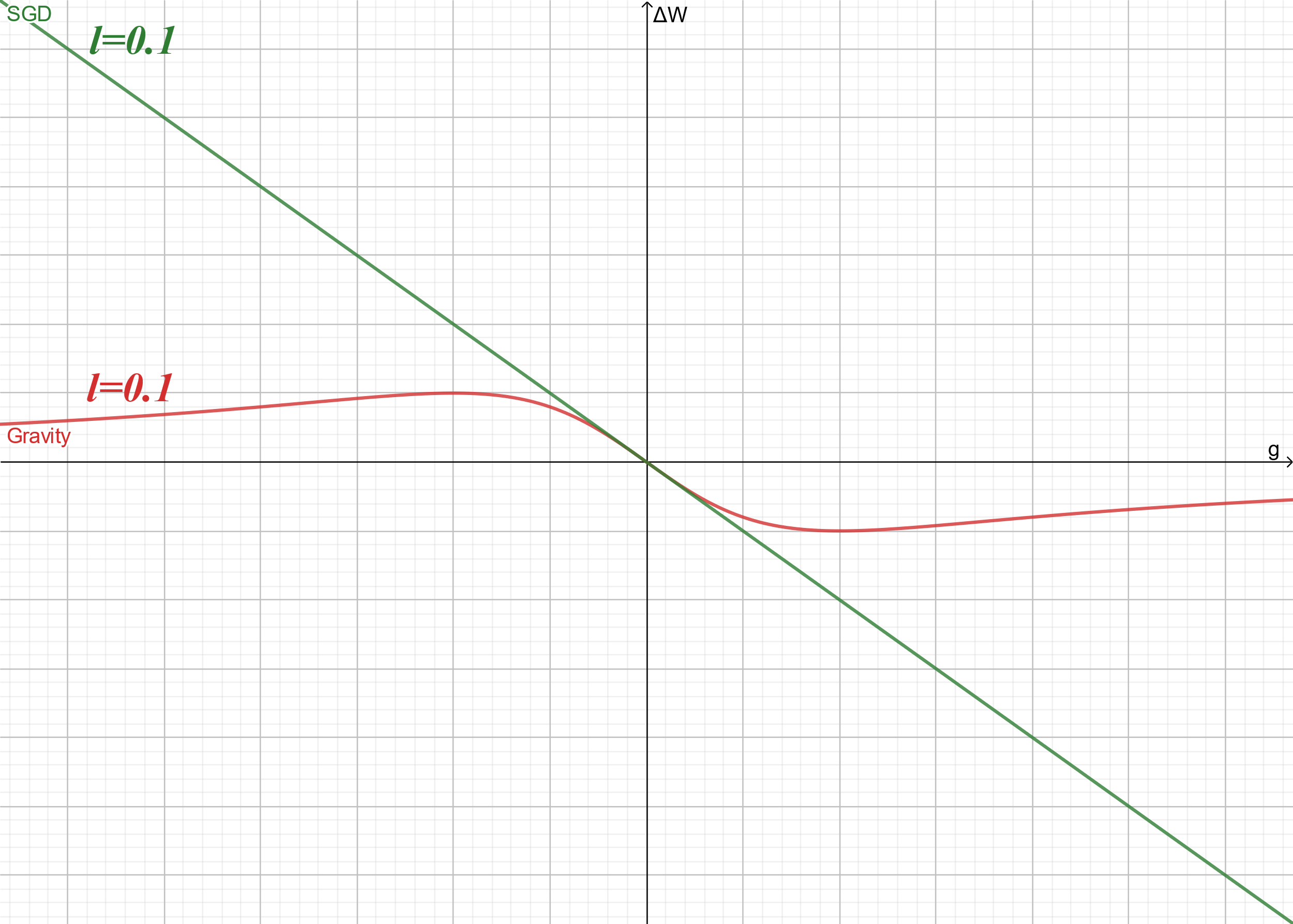


Figure 3. Gravity and SGD optimization function for learning rate value of 0.1

In this figure, it is clear that for small gradient values, gravity behaves like GSD. Mathematically: . To determine how small the gradient value must be for this similar behavior to occur, [Fig. 4](#Figure_4) is drawn for different values of the learning rate:

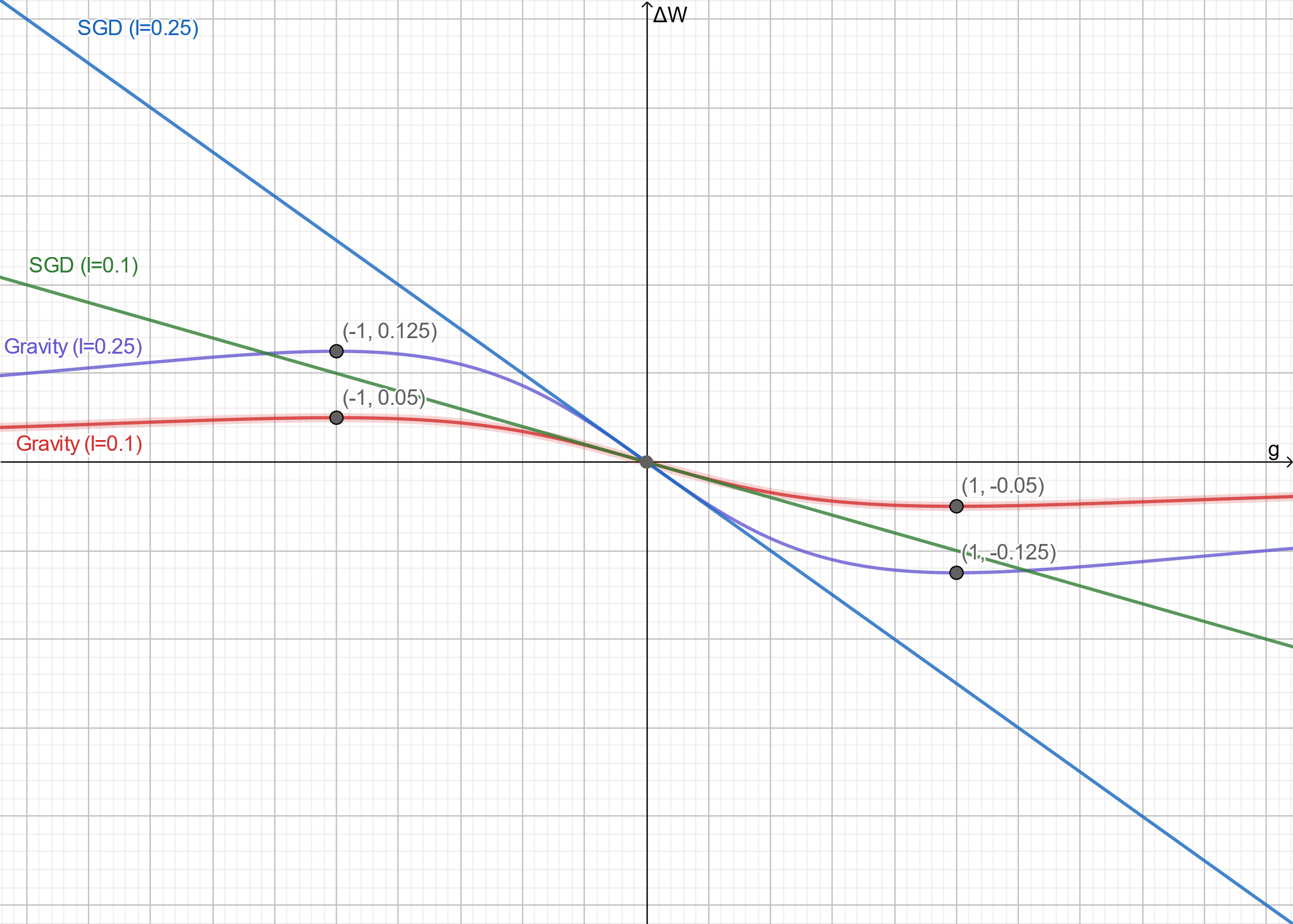


Figure 4. Gravity and SGD optimization function for learning rate values of 0.1 and 0.25

In this figure, it can be seen that by changing the learning rate, the gradient at which the maximum step is taken (maximum output value of the optimization function) does not change. Therefore, whatever learning rate is selected, the maximum step to change the parameter occurs at g = 1, which corresponds to 45 degrees.

## 2.2 Max-Step Grad

So far, we have transferred the two parameters we encountered in the Gravity optimization function ( and ) to learning rate hyper-parameter. The values ​​of these two parameters are the same for all weights of the weight matrix, i.e. they have the same values ​​for two different weights. But to gain more control over the optimization function, we need to have a parameter to change the slope angle. As we know, on sloping surfaces, by reducing the slope angle, we can watch two balls of different masses fall at a slower speed (such as a slow-motion video taken with a high-speed camera) and compare more easily. In contrast by increasing the slope angle, , the falling time will be reduced and everything happens quickly. Therefore, with more control over this angle, we can reach the minimum loss with more speed and less divergence. It can be done by tweaking gradient with a coefficient called Max-Step Grad, :

(15)

which . Plugging new from Eq. 15 into Eq. 8 and using Eq. 9 and Eq. 10 to simplify, new with more control is obtained as follows:

(16)

The learning rate for this more controlled optimization function is defined as:

(17)

Substituting Eq. 17 into Eq. 16 finally gives:

(18)

For a deeper understanding of Max-Step Grad hyper-parameter, Fig. 5 shows the effect of different values of this parameter, with the same values of learning rate, on the optimization function.

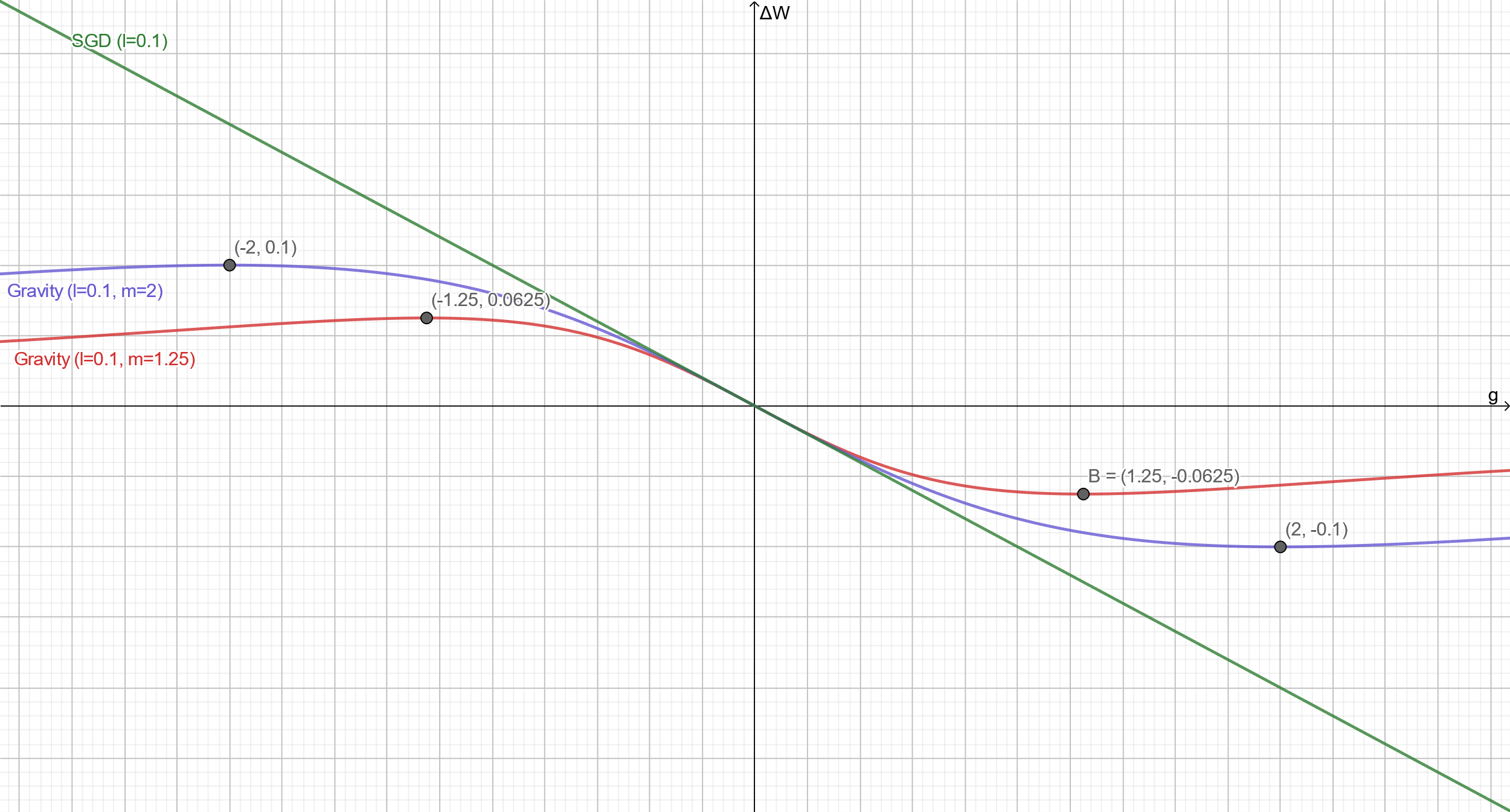


Figure 5. Gravity and SGD optimization function for Max-Step Grad values of 1.25 and 2 (learning rate = 0.1)

As mentioned at the beginning and shown in Figure 5, m is the gradient in which the most steps are taken to change the weight. Mathematically it can be said:

if and then .

Now the maximum step () for given Max-Step Grad () and learning rate () can be obtained as follows:

(19)

In summary, Max-Step Grad has two effects: the first is its effect on the linear region of the curve and the second is on the maximum step value. A greater value of Max-Step Grad results in a larger section of the linear region as well as bigger steps for weights with higher gradients. In other words, with increasing m, a wider range of the gradients is treated linearly, and also weights with larger gradient values take larger steps.

## 2.2.1 A Little About Gradient Descent Divergence

The reason for the divergence in Vanilla Gradient Descent for large learning rates is weights with large gradients. In fact in Gradient Descent optimization method, an infinite amount of is possible. Here is a scenario in gradient descent optimization method which leads to divergence:

1. Consider a weight with a large gradient.
2. This weight takes a bigger step relative to others (linearly proportional to their gradient ratio)
3. After applying optimizer, the weight goes too far and now has a larger gradient which leads to another big step
4. steps 1 to 3 will happen again to diverge.

As can be seen in Fig. 6, Harrington [[article](https://thelaziestprogrammer.com/sharrington/math-of-machine-learning/gradient-descent-learning-rate-too-high):] explains that more intuitively:

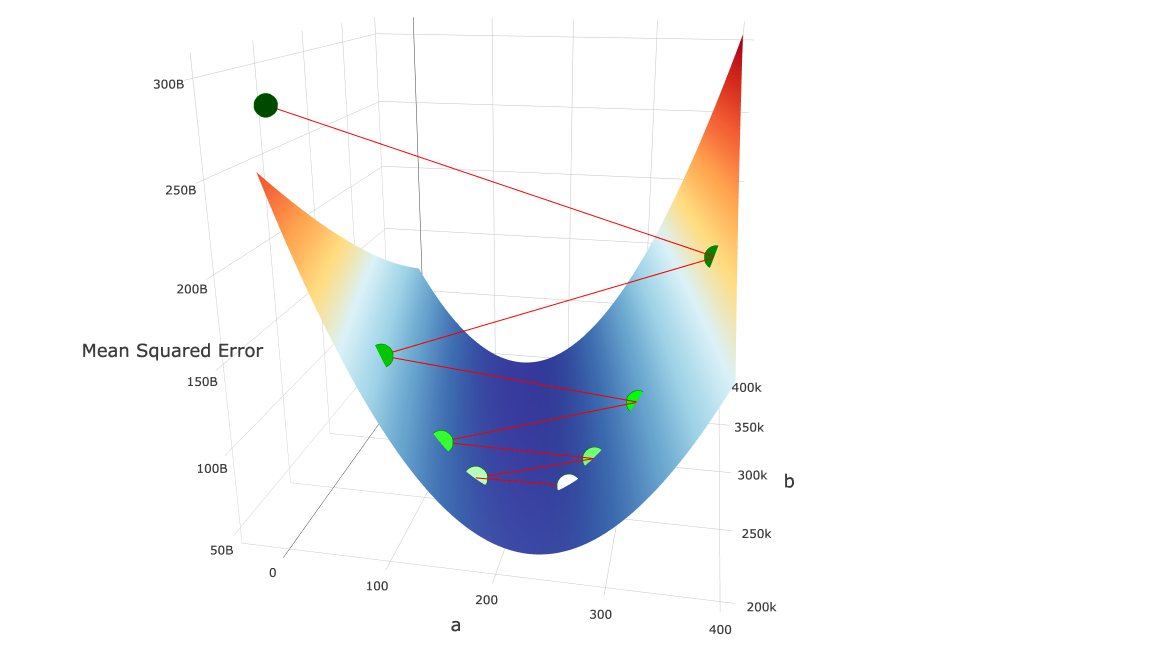


Figure 6. The illustration of divergence in gradient descent optimization method

1. We start at the white point in the “valley” and calculate the gradient at that point.
2. Then we multiply the learning rate by the gradient and move along this vector to the new point (the slightly greenish point to the left of the white point). Because the learning rate was so high, combined with the magnitude of the gradient, we “jumped over” the local minimum.
3. Then again we calculate the gradient at point 2, and make the next move. Again, we jump over the local minimum. The gradient at point 2 is even greater than the gradient at point 1. So in the following steps we again jump over the local minimum to diverge.
4. Due to the convex, valley-like curve of our objective function, as we continue to jump from side to side, the gradient at each jump grows higher. The error increases quadratically with each “jump”, and the algorithm diverges to infinite error.

## 2.3 Choosing M (Max-Step Grad)

we want to limit the for weights with larger . Given the fact that gradients are constantly changing during training, it is obvious that we cannot choose in advance. So we suggest selecting it based on the current gradient matrix. To avoid divergence, a gradient matrix with larger gradients must have a smaller . Based on this we suggest to select m as follows:

(20)

in this equation, G is the gradient matrix. Geometrical interpretation of this equation is as follow:

1. find the largest gradient or in other words steepest
2. calculate the complementary angle correspond to it:
3. and choose

## 2.4 Moving Average

Most of the common optimizers in deep learning like SGD with momentum, Adam, and RMSProp use moving average to stabilize loss reduction. Therefore we tested exponential moving average on Gravity optimizer and the results were promising. The main issue with Gravity before applying moving average was an initial delay in cost decay even though after some initial epochs without any loss reduction, the optimizer does its job very well. We first define gradient term, , as follow:

(21)

also, we define velocity, , as follow:

(22)

In Eq. 22, is a positive real number which . Also, is the value of in current update step (mini-batch) and is at the previous update step. The value of is initialized with 0 at . By these definitions new update rule based on is as follow:

(23)

The effect of on loss reduction is shown in Fig. 7 by comparison of different values of .

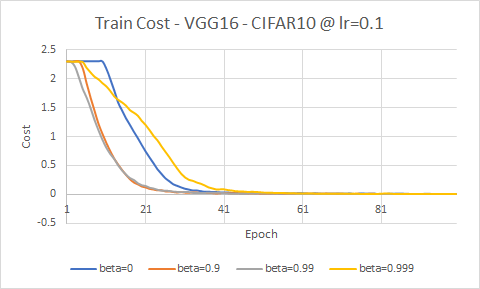


Figure 7. The effect of different values of on loss reduction

After performing some experiments that included changing the value of , we finally found that the optimal value for was 0.9 in most cases, although tuning was required in some special cases. While moving average helped Gravity with initial speed, there were still some delays. To solve this problem, we propose an alternative value for .

For any specific value of , the average is taken over a certain number of previous data. The number of these data can be calculated by Eq. 24 []. [equation](https://www.youtube.com/watch?v=NxTFlzBjS-4) []

(24)

In the first epochs, there is not enough data to be averaged and also at the beginning of the training which is small and , the value of Eq. 22 will be very small. There is a solution known as bias correction which modifies as follow:

(25)

The logic for Eq. 25 is that by increasing the value of , the denominator approaches to and Eq. 25 and Eq. 22 will output almost the same result. But at the beginning of the training, the value of is small so dividing Eq. 22 to this small value increases the total outcome.

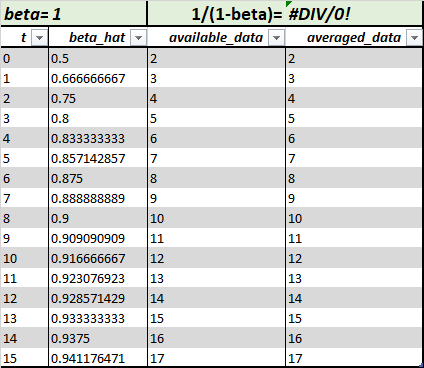
We tried to use bias correction but at large values of (closer to 1) we encounter overflow in our code. Therefore we tried to solve the problem of Eq. 22 at initial steps with a different approach. We propose an alternative to as follow:

(26)

Value of in Eq. 26 at large values of will tend to but at smaller values of will correct the value of . The derivation of Eq. 26 is described as follows:

Choosing any value for , the average will be over almost number of previous data. We define a variable value of which increases over time and tends to so the average will be always over all of the previous data. For averaging over all of the previous data in each step, the amount of data that will be averaged over should be ; because at , there are two data available, and . This can be written as follow:

At the value of is 0.5 which will average over two data ( and ). By increasing the value of , the value of tends to which will average over all of the previous data. Table shows the value of for different values of :



As can be seen in table , the number of data on which the average is done in each step is equal to the total number of available data. For averaging over number of previous data, the above equation is modified as which is Eq. 26. Now by increasing the value of , the value of tends to . The behavior of for different values of is shown in Fig. 8.

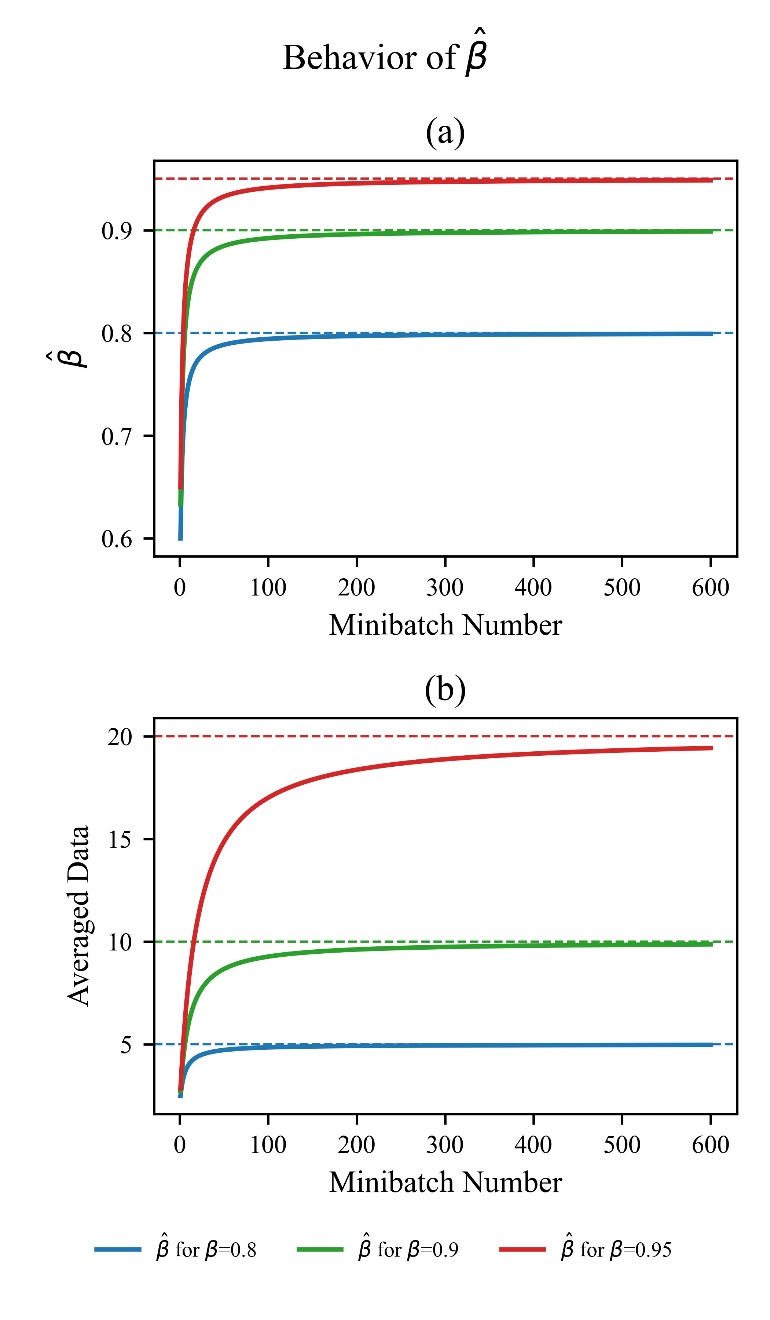


Figure 8 The behavior o of for different values of

In addition to modifying in Eq. 22, we came out with another solution for increasing the speed of Gravity optimizer at early steps by using non-zero initial . Instead of zero, was initialized with normally distributed numbers with a mean () of zero and a standard deviation of:

(27)

Take a closer look at the first update step:

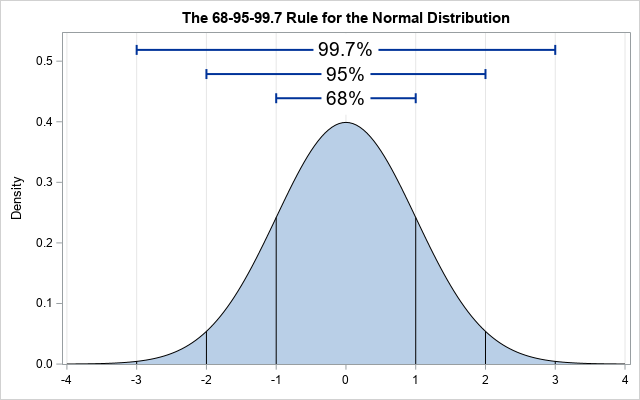
(28)

Eq. 28 can be seen as two separate parts. The first part is which is due to initial and the second part is which is due to the gradient term. For the second part, because it is about the gradient term and is determined by various parameters, nothing can be done. But the first part, , can be tweaked to reach faster initial loss reduction speed. This term is defined as follows:

(29)

If is initialized to zero, the total control of optimization will be at an unknown part of Eq. 28 which is .

For a normally distributed set of random numbers, 68% of values are less than the standard deviation (), 95% are less than , and 99.7% are less than . Therefore by choosing , the range of initial steps for different parameters (weights and biases) will be defined.



By selecting each sigma, 68% of the parameters will be as follows:

The numerator of the right side of this inequality is defined as a new hyper-parameter, called alpha , to give more control over the value of initial steps:

(30)

Eq. 30 is a different form of Eq. 27. Experiments with different values of alpha showed that is satisfying for most models.

## 2.5 Summary

In this section, a new optimizer based on back-propagation is proposed for deep learning from a kinematic point of view. The summary of its algorithm is:

**Require**: : Learning Rate *Recommended Value: 0.1*

**Require**: : Initial Step Size due to *Recommended Value: 0.05*

**Require**: : Moving Average Rate *Recommended Value: 0.9*

**Require**: : maximum number of update steps

for each parameter:

while

for each weight matrix :

*the gradient of objective function J w.r.t W*

note: *is element-wise division Hadamard division*

For easier implementation of the Gravity optimizer, its python implementation using TensorFlow’s high-level API, Keras, is available in Gravity GitHub repository.

# 3. Benchmark Configuration

In this section, we are going to compare Gravity optimizer with other common standard optimizers shown in Table 1. In the following, first, the specifications of the hardware that we used for training are given. Then the framework we used to implement the model, the datasets used for training, and finally, the architectures chosen based on hardware specifications are introduced. If you want to skip reading the details, a summary of this information is given in Table 4. In the last part, the obtained results from Gravity optimizer are compared to the reported results from other papers that have used the same architectures we used to train the same datasets we used. If results are not reported by other papers, the tests have been performed by us.

## 3.1 Hardware

We used [Google Colab](https://colab.research.google.com/) [14] as hardware because we couldn’t afford GPU for training deep neural network models and testing our ideas. We used TensorFlow’s high-level API, Keras, as a framework to build the model in the Python language. The python implementation code can be found in the [Gravity optimizer GitHub repository](https://github.com/dariush-bahrami/gravity.optimizer). Also, we were given the chance to use TPUs (Tensor Processing Unit: Google’s custom-developed technology to accelerate machine learning workloads) by using Google Colab and TensorFlow together.

## 3.2 Datasets

We used the following standard datasets to evaluate the performance of Gravity optimizer: MNIST, Fashion-MNIST, CIFAR-10, CIFAR-100 (Coarse), and CIFAR-100 (Fine). The MNIST database of handwritten digits is a subset of a larger set available from NIST. The images of digits have been size-normalized and centered in a fixed-size image [15]. The Fashion-MNIST is a dataset containing images of 10 classes. The 10 different classes are T-shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot[16]. CIFAR-10 is a subset of the 80 million tiny images dataset in 10 classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks [17]. CIFAR-100 is just like the CIFAR-10, except it has 100 classes containing 600 images each. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs). We trained the “fine” and “coarse” datasets separately because they have two distinct labels that classify two different types of classification; “coarse” is more general and “fine” is more specific[17]. Table 2 summarizes the detailed information of the datasets used.

Table 2. detailed information of datasets used for benchmark

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Dataset** | **Train #** | **Test #** | **Class #** | **Shape** | **Image per class** |
| MNIST [15] | 60 K | 10 K | 10 | 28x28x1 | 6 K |
| Fashion-MNIST [16] | 60 K | 10 K | 10 | 28x28x1 | 6 K |
| CIFAR-10 [17] | 50 K | 10 K | 10 | 32x32x3 | 5 K |
| CIFAR-100 (coarse) [17] | 50 K | 10 K | 20 | 32x32x3 | 2.5 K |
| CIFAR-100 (fine) [17] | 50 K | 10 K | 100 | 32x32x3 | 500 |

## 3.3 Architecture (models and hyper-parameters)

We used VGG16 and VGG19 with the exact specifications reported in their paper [18]. VGG16 has about 34M and VGG19 has about 39M parameters (detailed number of parameters for the input shape of 32x32x3 and 10 classes is shown in Table 3). Although architectures such as ResNet50[19] and EfficientNet[20] have 23M and 4M parameters respectively (for input shape of 32x32x3 and 10 classes; their model summary is available in [Gravity optimizer GitHub repository](https://github.com/dariush-bahrami/gravity.optimizer)) and they are as easy to implement as VGGNet in Keras, they showed so much slower training speed than VGGNet in Google Colab.

Optimization, regardless of its application in deep learning, is utilized to minimize a function. This action of minimization is the parameter that should be used for comparing the performance of optimizers. The function that is tried to be minimized in deep learning is the cost function. The parameter that should be used to compare optimizers in deep learning is the loss value in the training dataset. Therefore, to investigate the direct impact of the optimizer itself, we have using to use overfitting prevention techniques. Important examples of these techniques are learning rate decay[], dropout[], early stopping[], batch normalization[], and regularization[]. So another reason why we have chosen VGG architecture over other architectures is that it doesn’t use any overfitting prevention techniques.

Finally, we monitor loss and accuracy changes for training and validation datasets for a constant number of epochs (100 epochs) to compare the performance of Gravity optimizer with common standard optimizers listed in Table 1 (Where the same dataset and architecture is used without using overfitting prevention techniques. Table 3 summarizes the models used in this paper.

The remarkable thing about Gravity optimizer is that there was no need to tune hyper-parameters to get better results. The same values were considered in all benchmarks. In Section 2, we talked about why we designed them in that way and how to find the best values for them. Our recommended value for Gravity optimizer hyper-parameters was:

learning rate = 0.1 , Alpha = 0.01 , Beta = 0.9.

We also set these values as default for Gravity optimizer in python implementation. In this section, we use these suggested values for the benchmark.

To summarize, the results obtained from the training of five standard datasets mentioned in [section 3.2](#_3.2_Datasets) on VGGNet architectures (VGG16 and VGG19) using Gravity optimizers and two other standard and widely used optimizers (RMSProp and Adam) are compared in each subsection of datasets. As mentioned, all the training here is done with a batch size of 128 and for 100 epochs.

The activation function for all layers except the last layer is the ReLU function. It is defined as f(x) = max (0, x). As we know, the ReLU activation function was first used by Fukushima but not given any particular name [21]. Also Nair & Hinton's paper [22] spurred the recent interest in using the ReLU function in neural networks, and it is the source of the modern nomenclature “Rectified Linear Unit”.

In the TensorFlow documentation, it is strongly recommended not to use the Softmax function for multi-class classification and give logits (numeric output of the last linear layer of a multi-class classification neural network) directly to the cost function. Thus in the last layer, instead of using the Softmax function, classification is done by using Keras's sparse categorical cross-entropy class and turning the “from logits” attribute to True. Its python code is written as follows:

cost\_func = tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True)

In the following subsections, the results (last epoch and best epoch) were obtained from training our target datasets on VGG16 and VGG19 using Adam, RMSProp, and Gravity optimizers without using any overfitting prevention techniques are compared together. Also, the results are compared with the results reported from the other papers which used the same datasets and architectures we used here. More details of the results can be found in the [Gravity optimizer GitHub repository](https://github.com/dariush-bahrami/gravity.optimizer) or materials section.

Table 3. VGG16 and VGG19 model summary used in the paper

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **VGG16** | | | **VGG19** | | |
| **Layer Type** | **Output Size** | **Parameters#** | **Layer Type** | **Output Size** | **Parameters#** |
| **Convolution Part** | | | **Convolution Part** | | |
| Input Layer | 32, 32, 3 | 0 | Input Layer | 32, 32, 3 | 0 |
| Conv2D | 32, 32, 64 | 1,792 | Conv2D | 32, 32, 64 | 1,792 |
| Conv2D | 32, 32, 64 | 36,928 | Conv2D | 32, 32, 64 | 36,928 |
| MaxPooling2D | 16, 16, 64 | 0 | MaxPooling2D | 16, 16, 64 | 0 |
| Conv2D | 16, 16, 128 | 73,856 | Conv2D | 16, 16, 128 | 73,856 |
| Conv2D | 16, 16, 128 | 147,584 | Conv2D | 16, 16, 128 | 147,584 |
| MaxPooling2D | 8, 8, 128 | 0 | MaxPooling2D | 8, 8, 128 | 0 |
| Conv2D | 8, 8, 256 | 295,168 | Conv2D | 8, 8, 256 | 295,168 |
| Conv2D | 8, 8, 256 | 590,080 | Conv2D | 8, 8, 256 | 590,080 |
| Conv2D | 8, 8, 256 | 590,080 | Conv2D | 8, 8, 256 | 590,080 |
| MaxPooling2D | 4, 4, 256 | 0 | Conv2D | 8, 8, 256 | 590,080 |
| Conv2D | 4, 4, 512 | 1,180,160 | MaxPooling2D | 4, 4, 256 | 0 |
| Conv2D | 4, 4, 512 | 2,359,808 | Conv2D | 4, 4, 512 | 1,180,160 |
| Conv2D | 4, 4, 512 | 2,359,808 | Conv2D | 4, 4, 512 | 2,359,808 |
| MaxPooling2D | 2, 2, 512 | 0 | Conv2D | 4, 4, 512 | 2,359,808 |
| Conv2D | 2, 2, 512 | 2,359,808 | Conv2D | 4, 4, 512 | 2,359,808 |
| Conv2D | 2, 2, 512 | 2,359,808 | MaxPooling2D | 2, 2, 512 | 0 |
| Conv2D | 2, 2, 512 | 2,359,808 | Conv2D | 2, 2, 512 | 2,359,808 |
| MaxPooling2D | 1, 1, 512 | 0 | Conv2D | 2, 2, 512 | 2,359,808 |
| **Dense Part** | | | Conv2D | 2, 2, 512 | 2,359,808 |
| Flatten | 512 | 0 | Conv2D | 2, 2, 512 | 2,359,808 |
| Dense | 4096 | 2,101,248 | MaxPooling2D | 1, 1, 512 | 0 |
| Dense | 4096 | 16,781,312 | **Dense Part** | | |
| Dense | 10 | 40,970 | Flatten | 512 | 0 |
|  |  |  | Dense | 4096 | 2,101,248 |
|  |  |  | Dense | 4096 | 16,781,312 |
|  |  |  | Dense | 10 | 40,970 |
| **Total Parameters = 33,638,218** | | | **Total Parameters = 38,947,914** | | |

Table 4 shows a detailed summary of learning rate values used in runs. For Adam optimizer we turned off learning decay (decay = 0) and set beta 1 = 0.9, beta 2 = 0.999, and epsilon = 1.0 e-07. For RMSProp we turned learning rate decay, momentum, and centered off (decay = momentum = centered = 0) and set rho = 9.0 e-01, and epsilon = 1.0 e-07.

Table 4. summary of learning rates used for benchmark

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **VGG16** | | **VGG19** | |
|  | Adam | **RMSProp** | **Adam** | **RMSProp** |
| MNIST | 2.50e-04 | 1.00e-04 | 2.50e-04 | 2.50e-05 |
| Fashion-MNIST | 2.50e-04 | 5.00e-05 | 1.00e-05 | 5.00e-05 |
| CIFAR-10 | 1.00e-04 | 5.00e-05 | 1.00e-04 | 2.50e-05 |
| CIFAR-100 (coarse) | 1.00e-04 | 2.50e-04 | 7.50e-05 | 1.00e-04 |
| CIFAR-100 (fine) | 1.00e-04 | 1.00e-04 | 5.00e-05 | 1.00e-04 |

# 4. Results

In this section, the results obtained from the training of selected datasets on two architectures (VGG16 and VGG19) using three different optimization techniques (Adam, RMSProp, and our proposed optimizer) are reported. At the end of the previous section and also in Table 4, detailed information of the hyper-parameters used for all three optimizers is given.

## 4.1 MNIST Results

In this subsection, the MNIST dataset is trained first on VGG16 architecture and then on VGG19 architecture without using any overfitting prevention techniques (we discussed the reason for not using overfitting prevention techniques in [section 3.3](#_3.3_Architecture_(models)). To compare the results obtained from all three optimization techniques, their results are given in Figures 1 and 2.

### 4.1.1 MNIST Results on VGG16

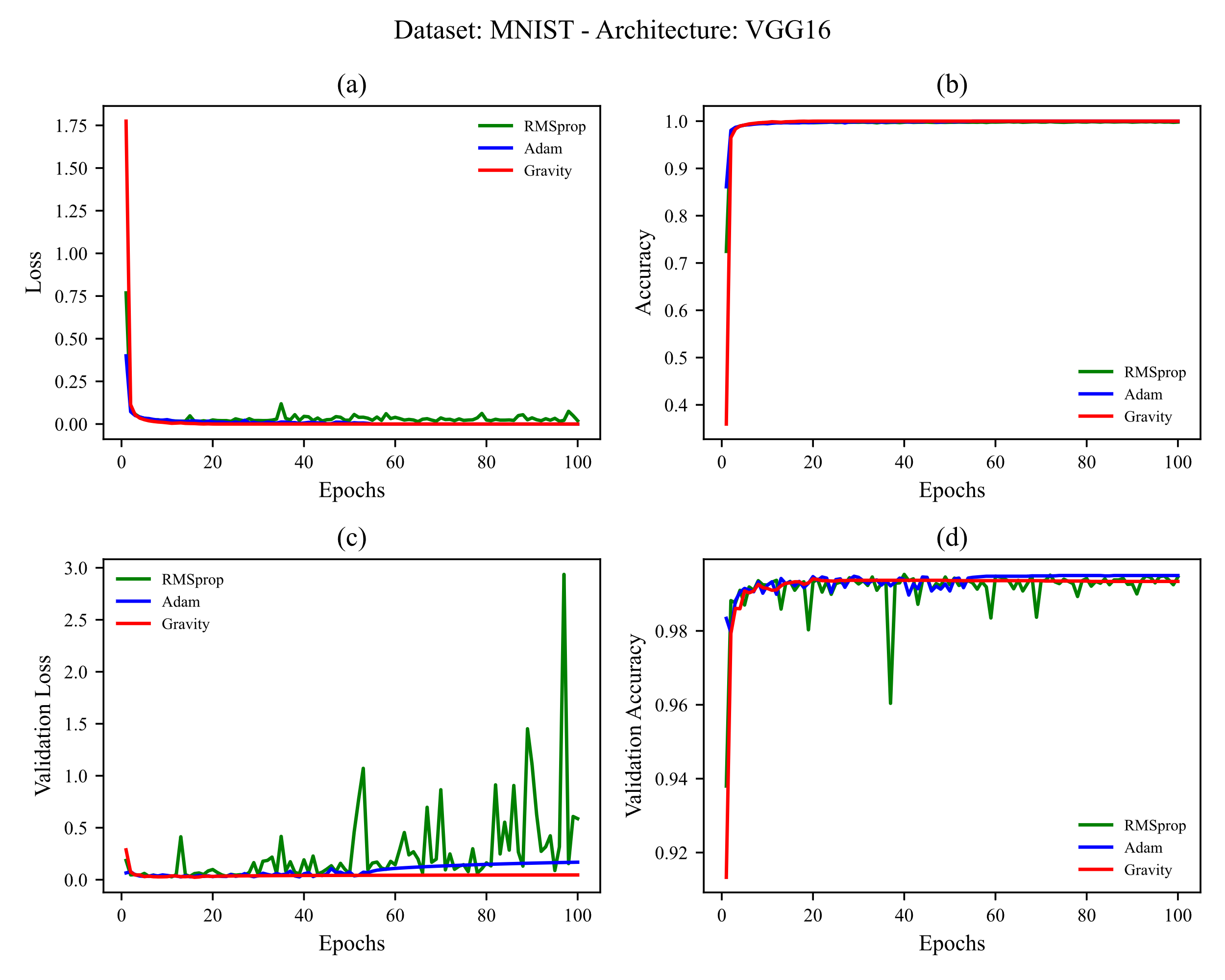


Figure 9. training MNIST on VGG16. learning rates used are shown in Table 4

### 4.1.2 MNIST Results on VGG19

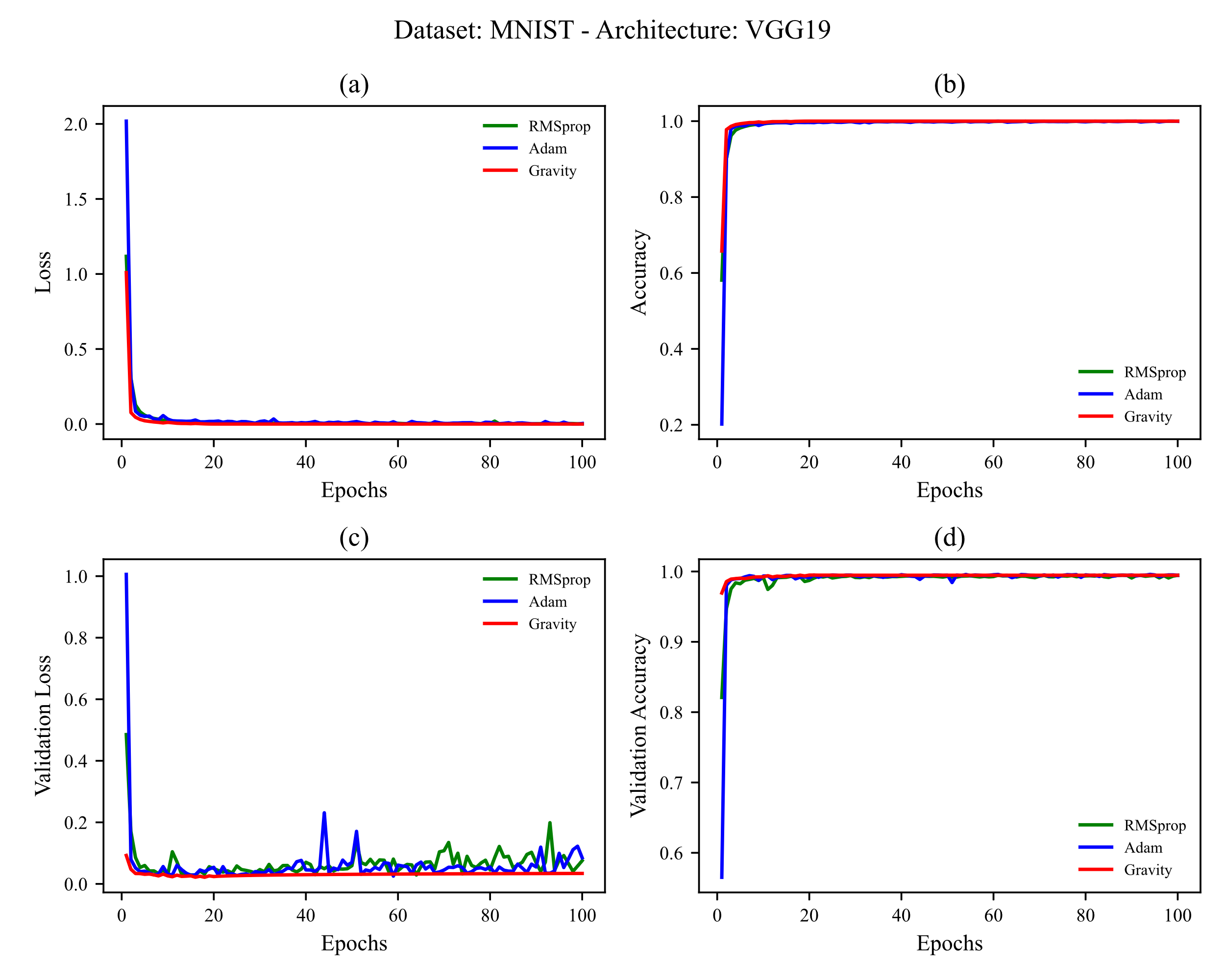


Figure 10 training MNIST on VGG19. learning rates used are shown in Table 4

## 4.2 Fashion-MNIST

In this subsection, the Fashion-MNIST dataset is trained first on VGG16 architecture and then on VGG19 architecture without using any overfitting prevention techniques (we discussed the reason for not using overfitting prevention techniques in [section 3.3](#_3.3_Architecture_(models)). To compare the results obtained from all three optimization techniques, their results are given in Figures 3 and 4.

### 4.2.1 Fashion-MNIST on VGG16

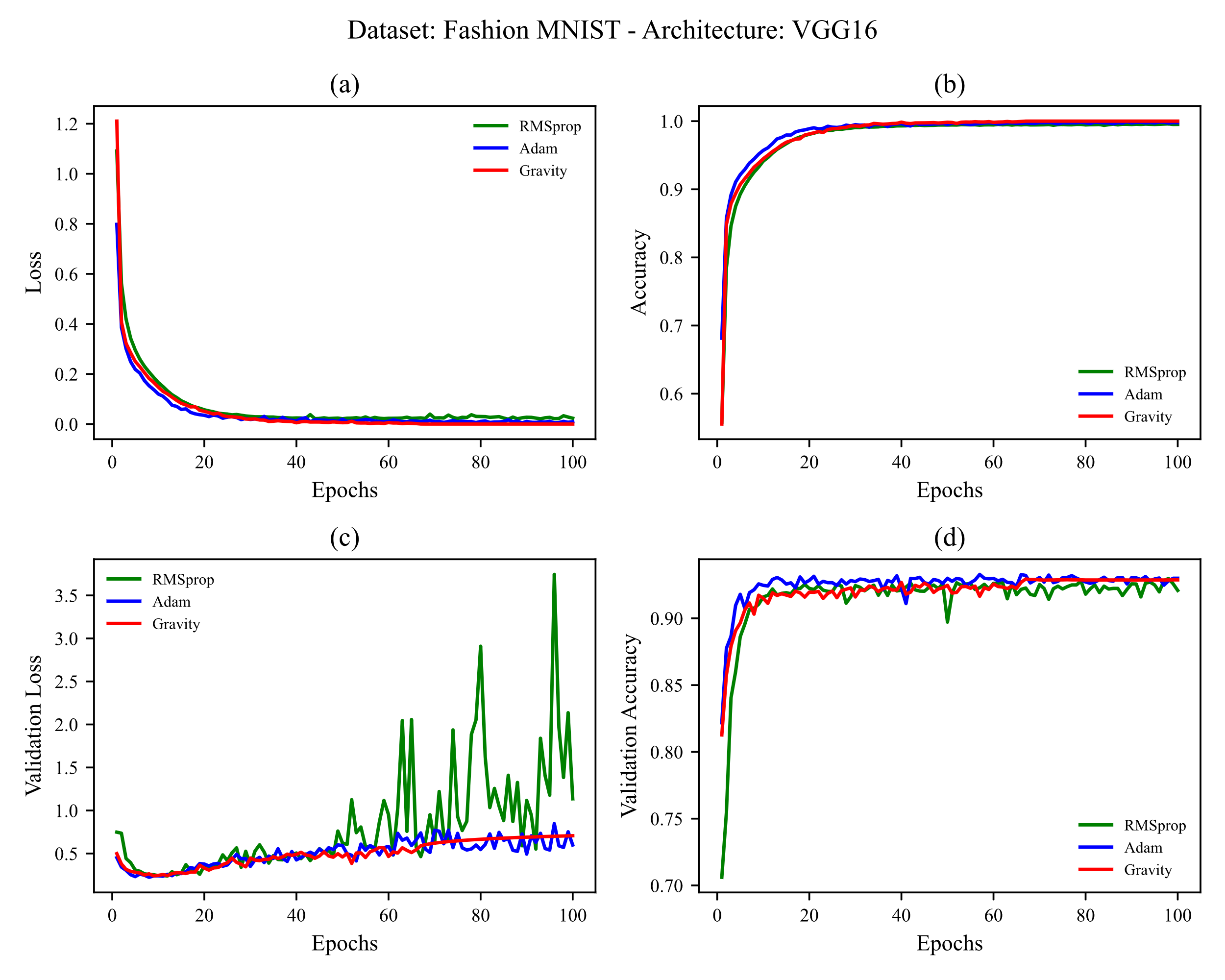


Figure 11 training Fashion-MNIST on VGG16. learning rates used are shown in Table 4

### 4.2.2 Fashion-MNIST on VGG19

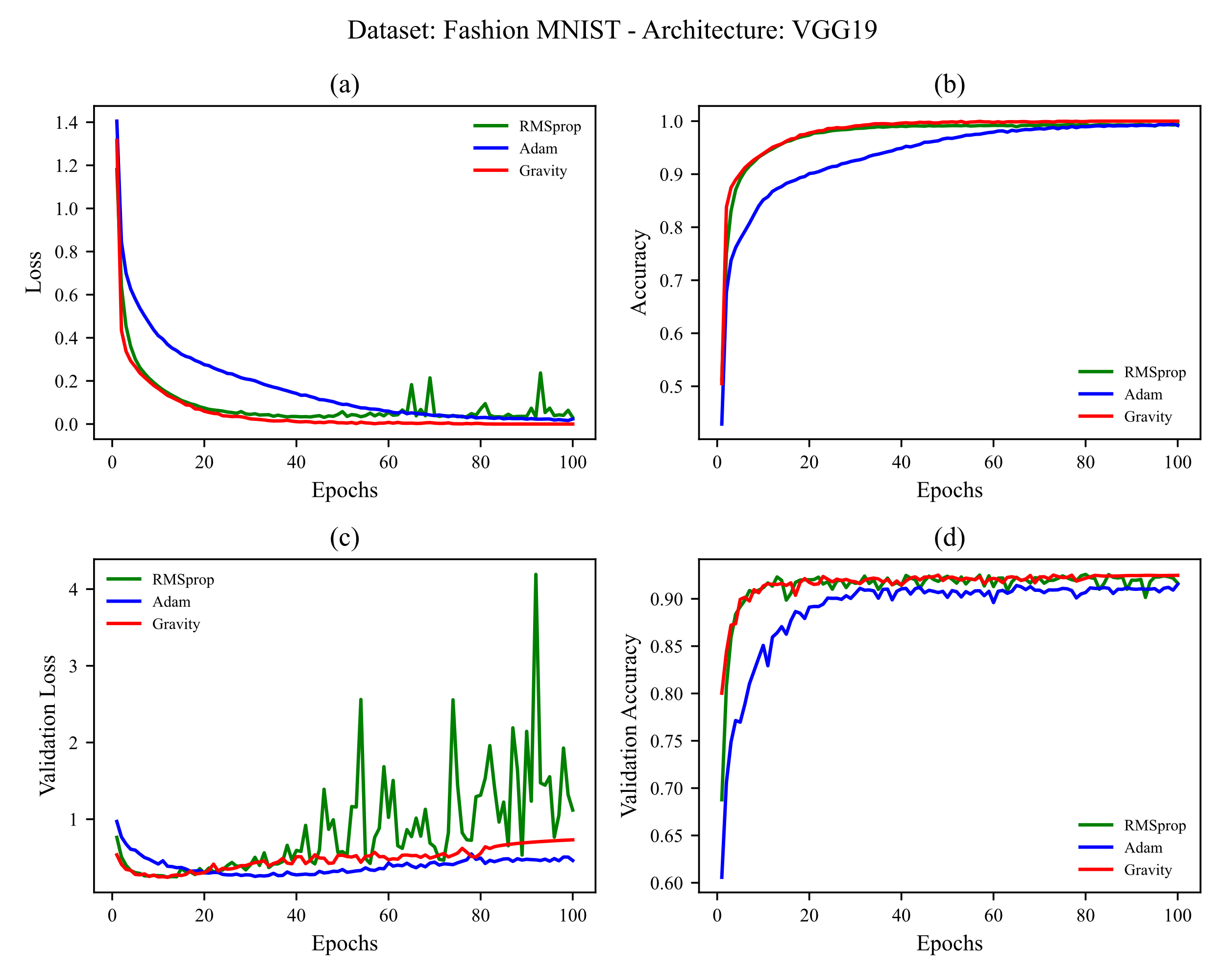


Figure 12 training Fashion-MNIST on VGG19. learning rates used are shown in Table 4

## 4.3 CIFAR-10

In this subsection, the CIDAR-10 dataset is trained first on VGG16 architecture and then on VGG19 architecture without using any overfitting prevention techniques (we discussed the reason for not using overfitting prevention techniques in [section 3.3](#_3.3_Architecture_(models)). To compare the results obtained from all three optimization techniques, their results are given in Figures 5 and 6.

### 4.3.1 CIFAR-10 on VGG16

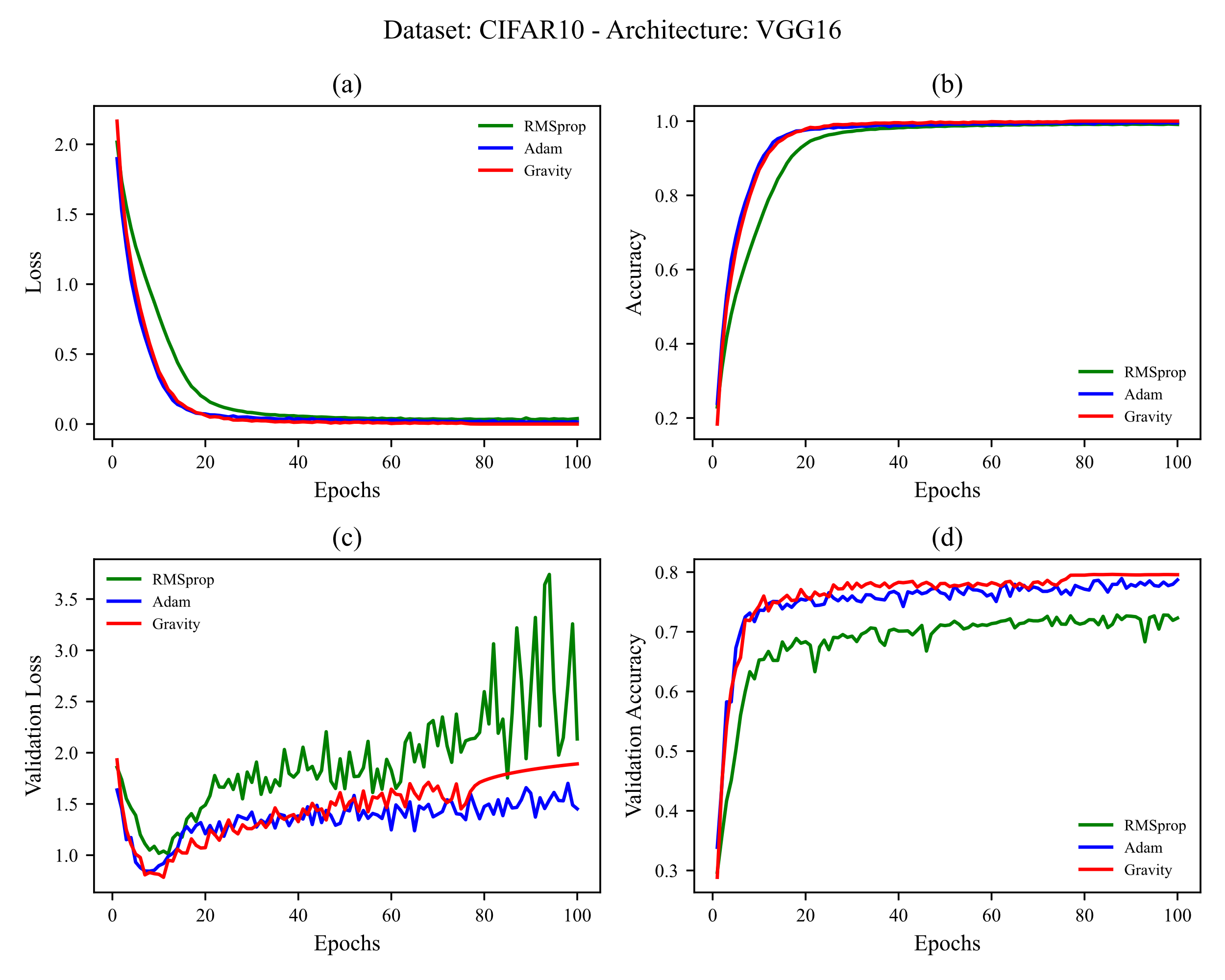


Figure 13 training CIFAR-10 on VGG16. learning rates used are shown in Table 4

### 4.3.1 CIFAR-10 on VGG19

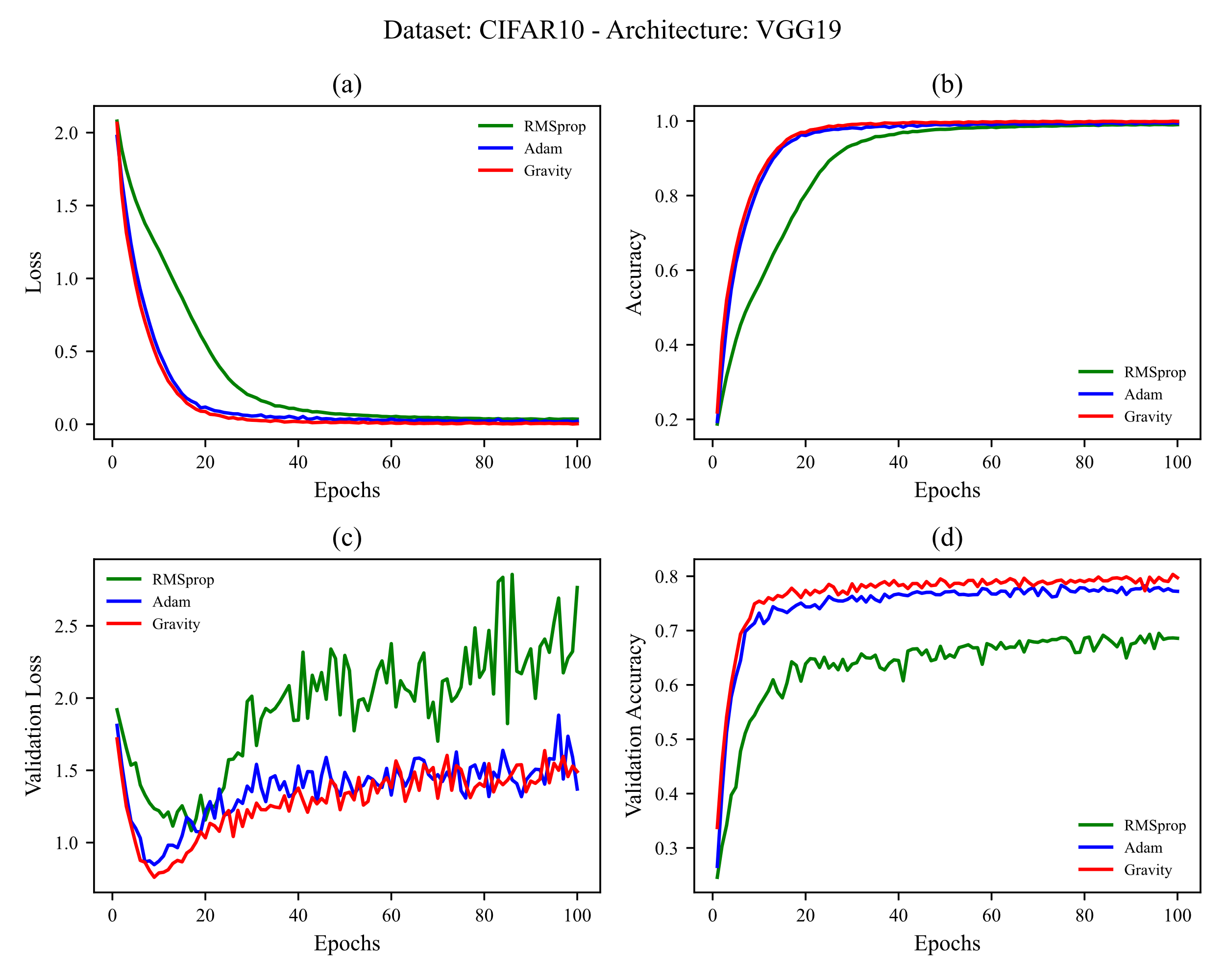


Figure 14 training CIDAR-10 on VGG19. learning rates used are shown in Table 4

## 4.4 CIFAR-100 (Coarse)

In this subsection, the CIFAR-100 (Coarse) dataset is trained first on VGG16 architecture and then on VGG19 architecture without using any overfitting prevention techniques (we discussed the reason for not using overfitting prevention techniques in [section 3.3](#_3.3_Architecture_(models)). To compare the results obtained from all three optimization techniques, their results are given in Figures 7 and 8.

### 4.4.1 CIFAR-100 (Coarse) on VGG16

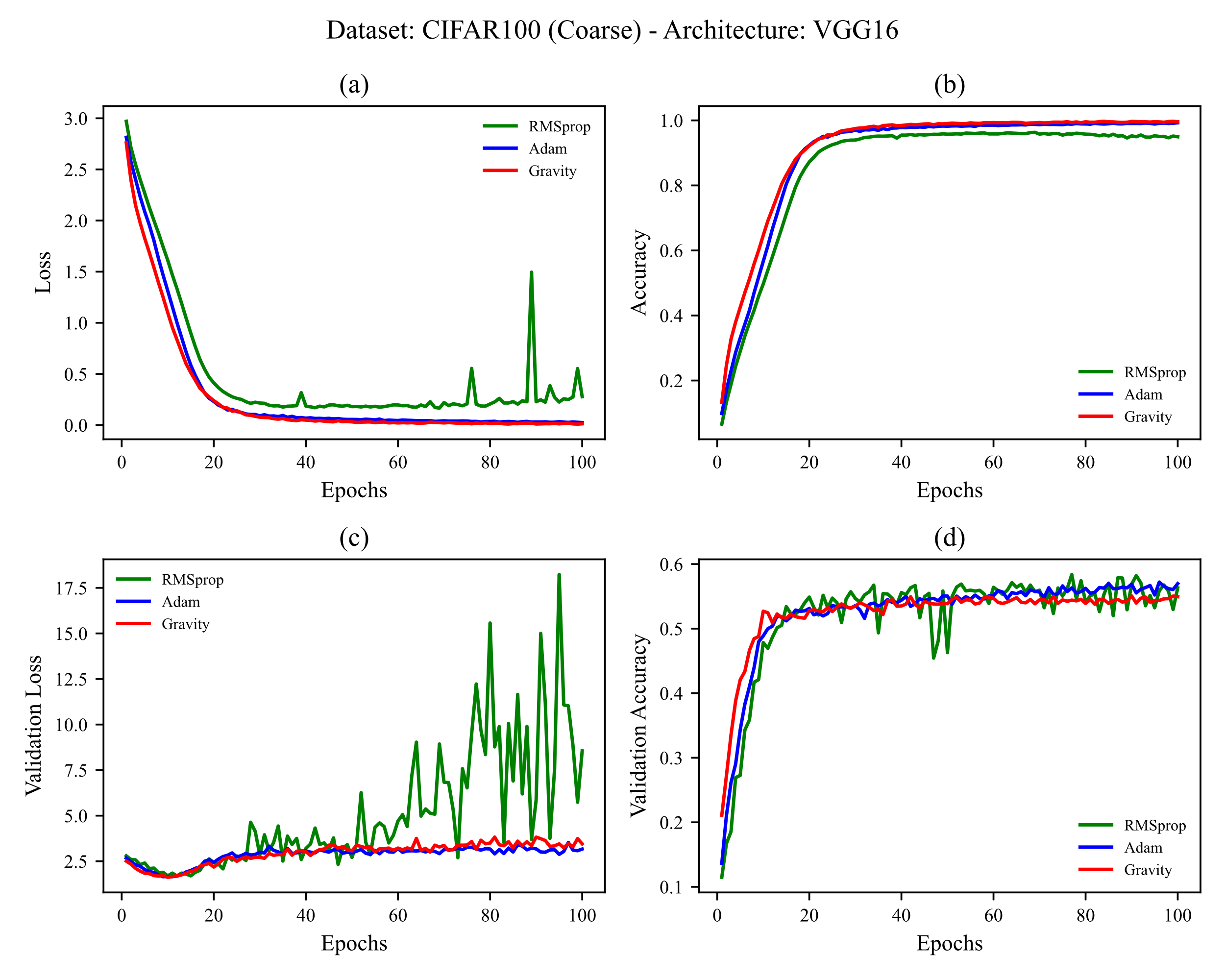


Figure 15 training CIFAR-100 (Coarse) on VGG16. learning rates used are shown in Table 4

### 4.4.2 CIFAR-100 (Coarse) on VGG19

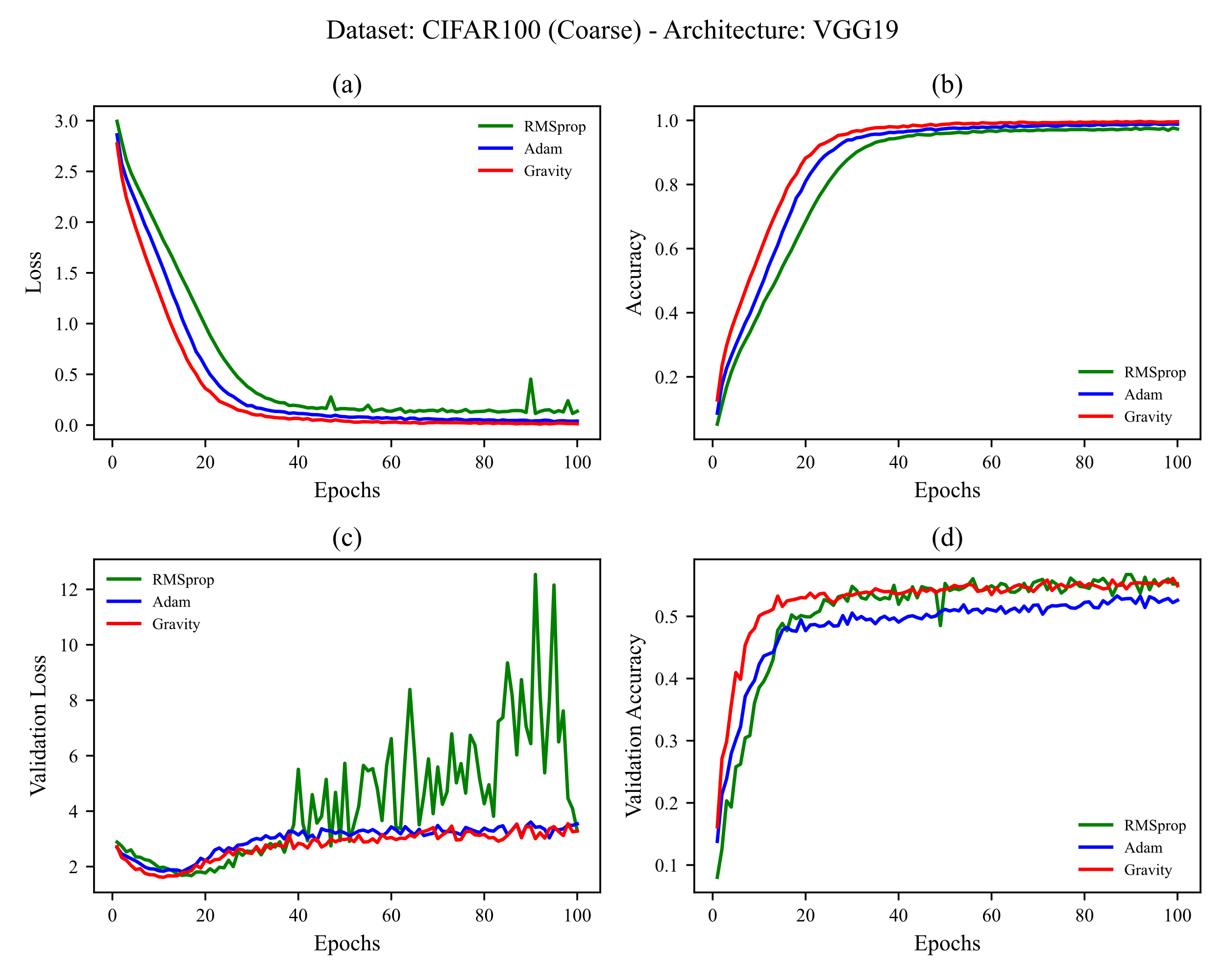


Figure 16 training CIFAR-100 (Coarse) on VGG19. learning rates used are shown in Table 4

## 4.5 CIFAR-100 (Fine)

In this subsection, the CIFAR-100 (Fine) dataset is trained first on VGG16 architecture and then on VGG19 architecture without using any overfitting prevention techniques (we discussed the reason for not using overfitting prevention techniques in [section 3.3](#_3.3_Architecture_(models)). To compare the results obtained from all three optimization techniques, their results are given in Figures 9 and 10.

### 4.5.1 CIFAR-100 (Fine) on VGG16

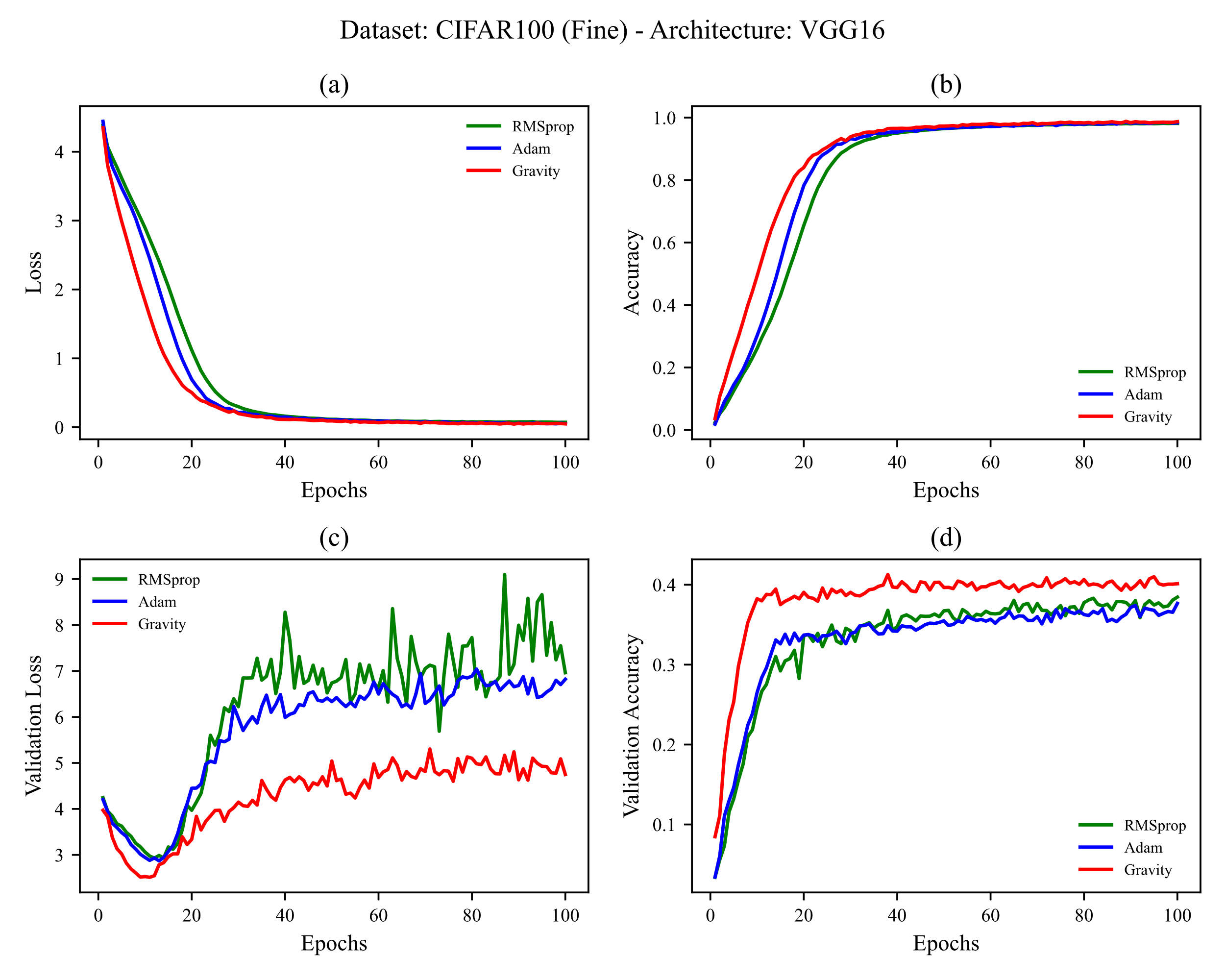


Figure 17 training CIFAR-100 (Fine) on VGG16. learning rates used are shown in Table 4

### 4.5.2 CIFAR-100 (Fine) on VGG19

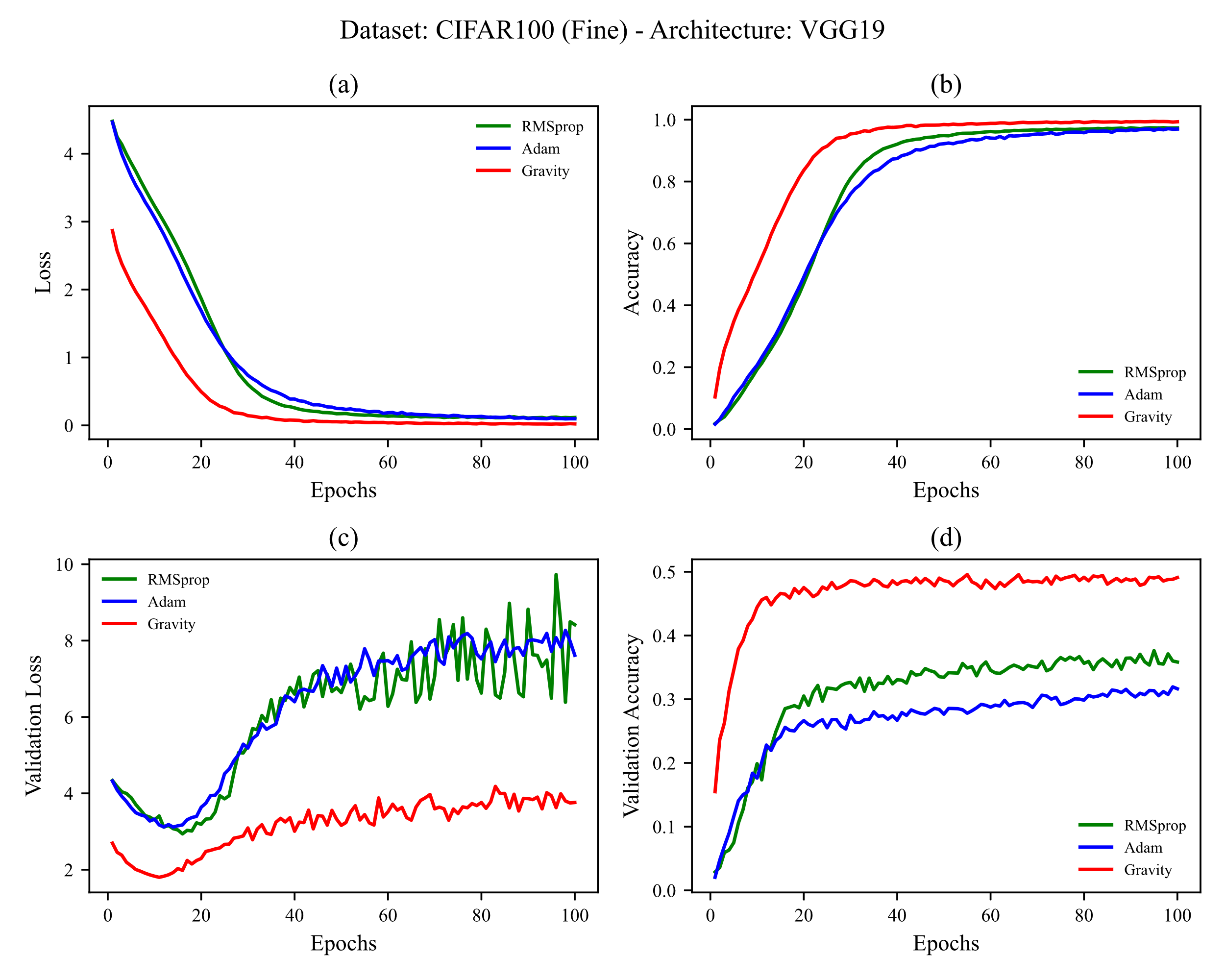


Figure 18 training CIFAR-100 (Fine) on VGG19. learning rates used are shown in Table 4

# 5. Conclusion

**# TODO: نوشتن جمع بندی تو مرحله آخر**

# References

[1] D. Choi, C.J. Shallue, Z. Nado, J. Lee, C.J. Maddison, G.E. Dahl, On empirical comparisons of optimizers for deep learning, ArXiv Prepr. ArXiv1910.05446. (2019).

[2] D. Soydaner, A Comparison of Optimization Algorithms for Deep Learning, Int. J. Pattern Recognit. Artif. Intell. (2020) 2052013.

[3] A.C. Wilson, R. Roelofs, M. Stern, N. Srebro, B. Recht, The marginal value of adaptive gradient methods in machine learning, in: Adv. Neural Inf. Process. Syst., 2017: pp. 4148–4158.

[4] F. Schneider, L. Balles, P. Hennig, DeepOBS: A deep learning optimizer benchmark suite, ArXiv Prepr. ArXiv1903.05499. (2019).

[5] H. Robbins, S. Monro, A stochastic approximation method, Ann. Math. Stat. (1951) 400–407.

[6] G. Hinton, N. Srivastava, K. Swersky, Neural networks for machine learning, Coursera, Video Lect. 264 (2012).

[7] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, ArXiv Prepr. ArXiv1412.6980. (2014).

[8] B.T. Polyak, Some methods of speeding up the convergence of iteration methods, USSR Comput. Math. Math. Phys. 4 (1964) 1–17.

[9] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization., J. Mach. Learn. Res. 12 (2011).

[10] M.D. Zeiler, Adadelta: an adaptive learning rate method, ArXiv Prepr. ArXiv1212.5701. (2012).

[11] I. Sutskever, J. Martens, G. Dahl, G. Hinton, On the importance of initialization and momentum in deep learning, in: Int. Conf. Mach. Learn., 2013: pp. 1139–1147.

[12] T. Dozat, Incorporating nesterov momentum into adam, (2016).

[13] J.R. Sashank, K. Satyen, K. Sanjiv, On the convergence of adam and beyond, in: Int. Conf. Learn. Represent., 2018.

[14] E. Bisong, Google Colaboratory, in: Build. Mach. Learn. Deep Learn. Model. Google Cloud Platf., Springer, 2019: pp. 59–64.

[15] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE. 86 (1998) 2278–2324.

[16] H. Xiao, K. Rasul, R. Vollgraf, Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, ArXiv Prepr. ArXiv1708.07747. (2017).

[17] A. Krizhevsky, G. Hinton, others, Learning multiple layers of features from tiny images, (2009).

[18] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, ArXiv Prepr. ArXiv1409.1556. (2014).

[19] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2016: pp. 770–778.

[20] M. Tan, Q. V Le, Efficientnet: Rethinking model scaling for convolutional neural networks, ArXiv Prepr. ArXiv1905.11946. (2019).

[21] K. Fukushima, S. Miyake, Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition, in: Compet. Coop. Neural Nets, Springer, 1982: pp. 267–285.

[22] V. Nair, G.E. Hinton, Rectified linear units improve restricted Boltzmann machines, in: ICML, 2010.