**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. the most common standard optimizers in deep learning.

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 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 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 and overfitting techniques) are reported in detail. In Section 4, 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 summary of the results obtained and what needs to be done in the future on the gravity optimizer.

# 2 Gravity Optimizer

Let’s imagine an inclined plane and write some basic kinematic physics for that:

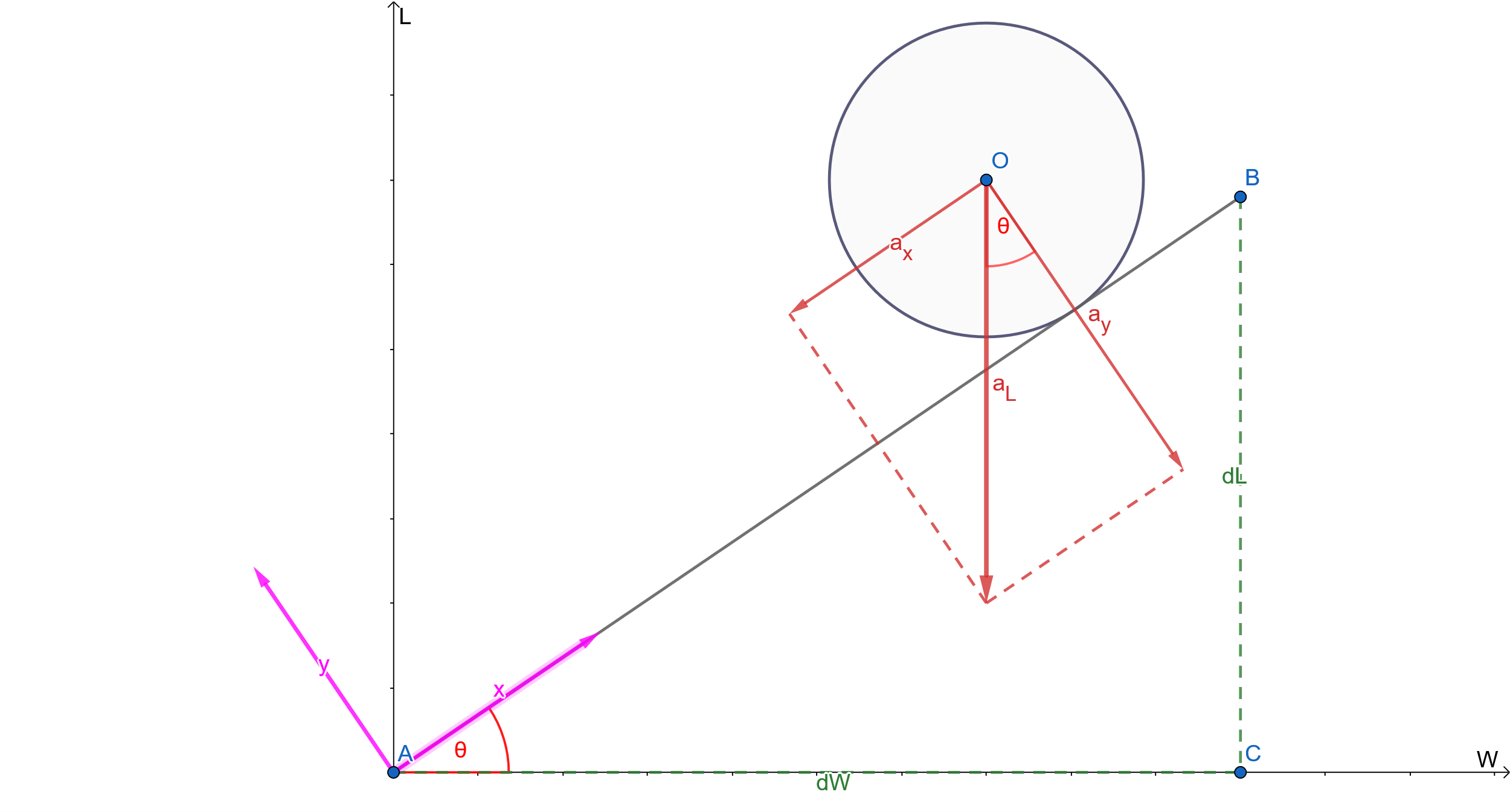


Figure . caption to be written

In this simple example let’s (we/nothing!) forget the rolling part of the ball and focus on , the acceleration along the L axis. First, we calculate by using and :

(1)

From now on we call as which stands for the gradient.

In this imaginary world, there is a universal acceleration in the L axis direction which is . Let’s calculate the relationship between the inclined plane's local coordinate system which is and the global coordinate system which is By using basic trigonometric relationships we can achieve this as follow:

(2)

(3)

Now let’s infer :

(4)

In this situation that and does not change with time, we can write a constant acceleration equation for the position as follow:

(5)

by assuming , the new shape of equation 5 will be as below:

(6)

and by substituting equation 6 into equation 2 we can write:

(7)

Plug from Equation 4 in Equation 7:

(8)

[equations](https://en.wikipedia.org/wiki/Inverse_trigonometric_functions) for calculating sine and cosine of are as follow:

(9)

(10)

Now by using equations 1, 8, 9, 10 we can write:

(11)

Equation 11 is our raw parameter-update equation. but as you can see there is a lot of hyper-parameters in this equation that need a lot of time for tuning. besides that, these hyper-parameters are not intuitive.

## 2.1 Learning Rate

we used to work with more familiar hyper-parameters like learning rate. so let’s wrap up every hyper-parameter until now as a single one which is learning rate:

(12)

then:

(13)

this is a more intuitive equation but it does not look so reasonable! how we can be sure about equation 12? is that a learning rate or just a name for an unknown hyper-parameter?

let’s look at optimization methods in a different way. we can simplify all back-prop based optimization methods as a function of the gradient. every optimization method gets a gradient and calculates a step for the parameter to take. for example, vanilla Gradient Descent is a linear function of the gradient:

(14)

we can plot that as below:

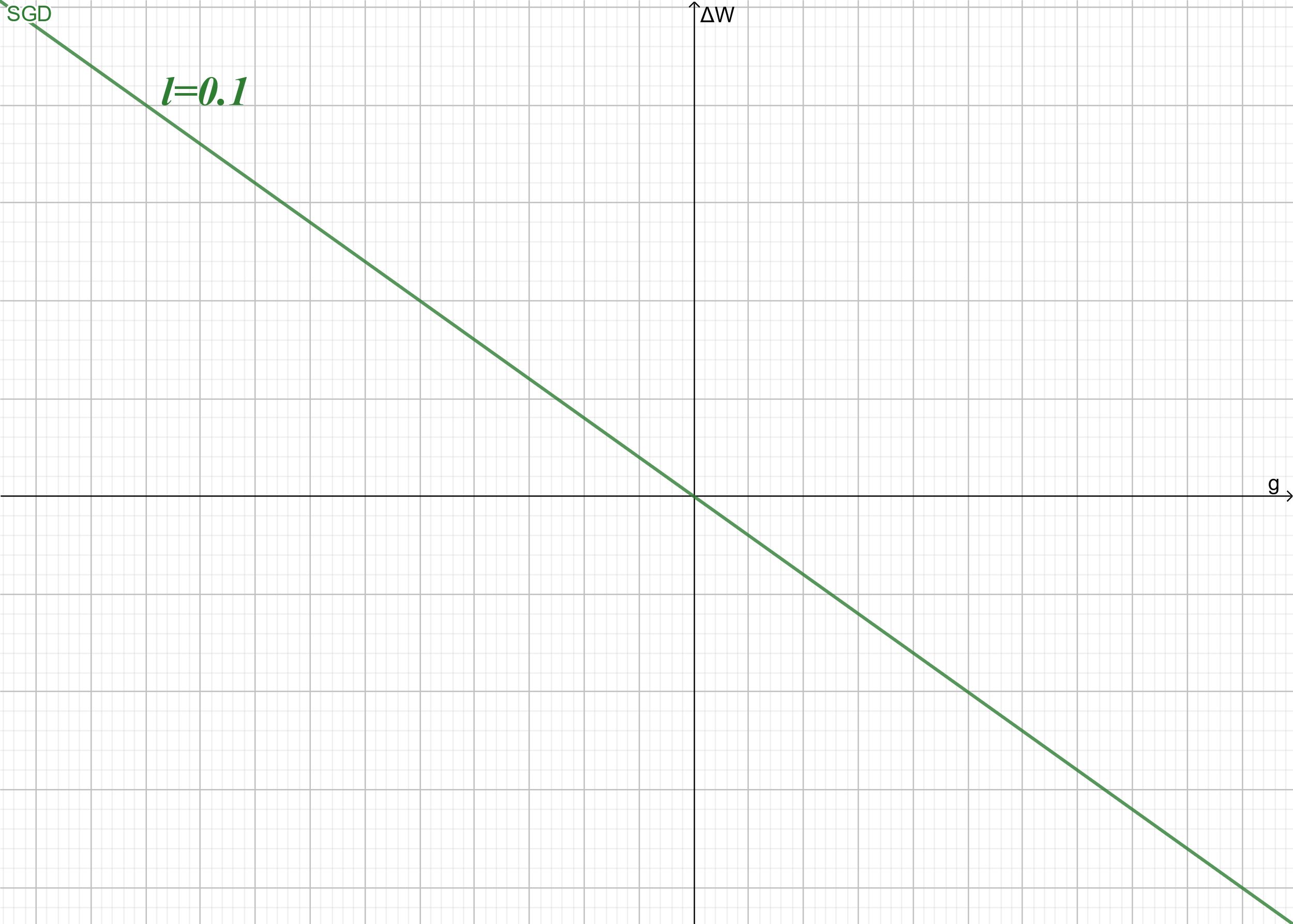


Figure . The caption to be written

it is obvious from this plot that for a small value of gradient, Gravity behave like Gradient Descent. in math words . but how small? let's try different learning rate and plot that too:

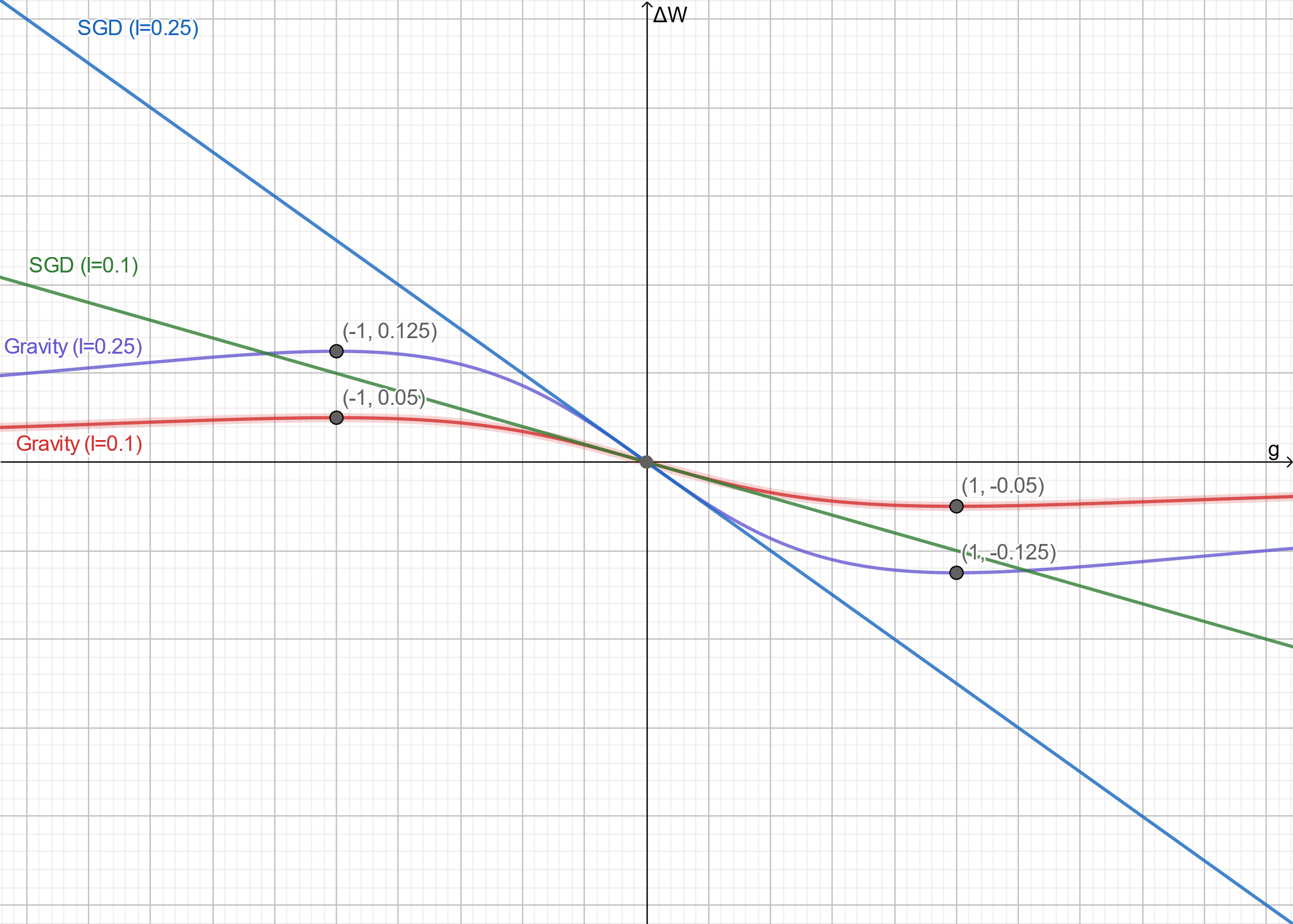


Figure . caption to be written

in this plot, we can see by changing the learning rate the gradient at which the maximum step occurs(i.e. Extremum) will not change. no matter what learning rate we choose, the maximum step always occurs at $g=1$ which corresponds to 45°.

## 2.2 Max-Step Grad

We need more control. so far only parameters we encounter are and which both of them are universal. by universal we mean they are the same for every parameter in the weight matrix. we want specific control on this particular parameter .well, say hello to Galileo Galilei.

you probably heard the myth which states that Galileo had dropped balls from the Leaning Tower of Pisa to demonstrate that their time of descent was independent of their mass. While this story has been retold in popular accounts, there is no account by Galileo himself of such an experiment, and it is generally accepted by historians that it was at most a thought experiment that did not take place. However, most of his experiments with falling bodies were carried out using inclined planes where both the issues of timing and air resistance were much reduced. inclined plane acts like a slow-motion video recorded by a high-speed camera. when you increase the angle the falling time will be reduced and everything happens quickly. in contrast by reducing we bend time and slow down the ball's falling motion.

we can do what Galileo did by changing in a way that helps us to reach the minimum of more quickly and without divergence. we will do this by tweaking gradient with a coefficient called $m$ which . let’s change equations that deal with g:

(15)

(16)

(17)

(18)

let’s look at the effect of m in the plot:

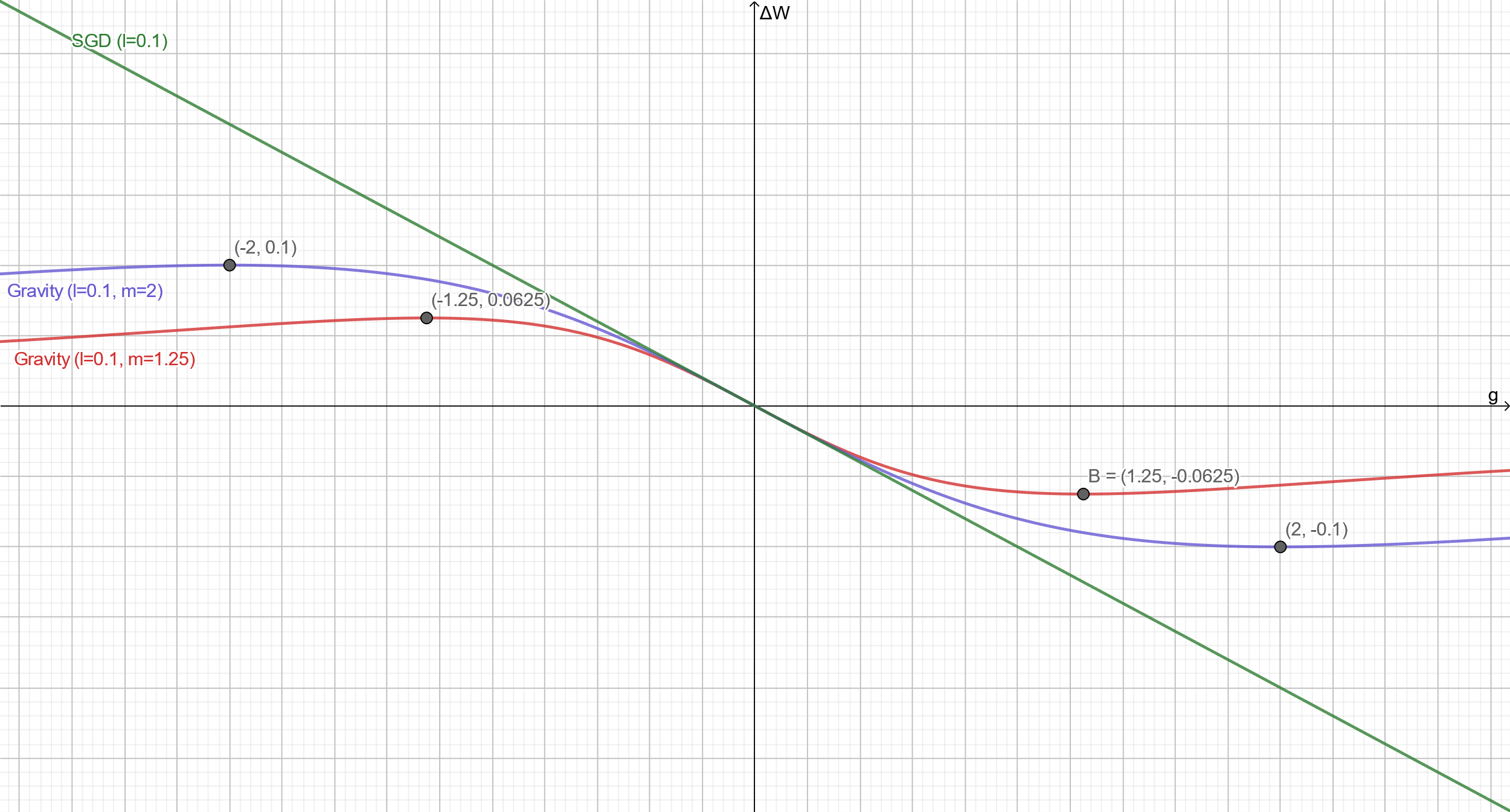


Figure . caption to be added

as can be seen, m is gradient which at that, the maximum step will occur. in math language if and then .

this parameter enables us to tweak the angle of the inclined plane for our benefit. the maximum step for given and will be:

(19)

has two effects; the first one is its effect on the linear part of the curve and the second one is the maximum step value. higher leads to a wider linear part and also a bigger step for weights with big . in other words by increasing a wider range of gradients will be treated linearly and weights with larger gradient value will take larger steps.

## 2.2.1 A Little About Gradient Descent Divergence

the cause of divergence in vanilla gradient descent at larger learning rates is weights with large gradients. in fact in the gradient descent optimization method, an infinite amount of is possible! one common scenario in gradient descent divergence is as follow:

1. consider a wight 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 forever

Sean Harrington in his awesome [article](https://thelaziestprogrammer.com/sharrington/math-of-machine-learning/gradient-descent-learning-rate-too-high) explains this more intuitively:

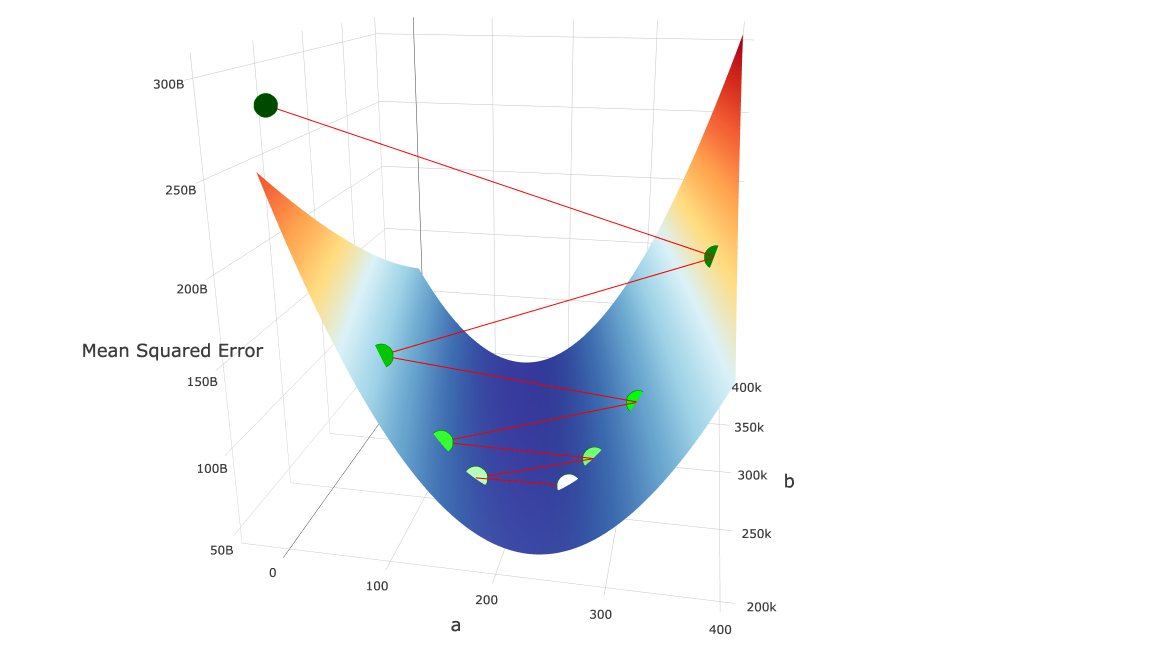


Figure 5

1. We start at the white point in the “valley” and calculate the gradient at that point.
2. We multiply our learning rates by our gradient and move along this vector to our new point (the slightly greenish point to the left of the white point) Because our learning rate was so high, combined with the magnitude of the gradient, we “jumped over” our local minimum.
3. We calculate our gradient at point 2, and make our next move, again, jumping over our local minimum

\* Our gradient at point 2 is even greater than the gradient at point 1!

\* Our next step will again, jump over our valley, and we will rinse and repeat for eternity.

1. 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. Our error increases quadratically with each “jump”, and our 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 that you select m as follows:

(20)

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

1. we found the largest gradient and therefore steepest
2. calculate the complementary angle correspond to it:
3. and choose

## 2.3 Moving Average

Most of the common optimizers in deep learning like momentum, Adam, and RMSProp use moving average to stabilize loss reduction therefore we tested exponential moving average on gravity and the results were promising. the main issue with gravity before applying moving average was an initial delay in cost decay although 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, V, as follow:

(22)

in equation 21 is a positive real number which . is value in current update step (mini-batch) and is at the previous update step and we initial with 0 at . by these definitions new update rule based on is as follow:

(23)

The below figure shows the effect of different values of on loss reduction:

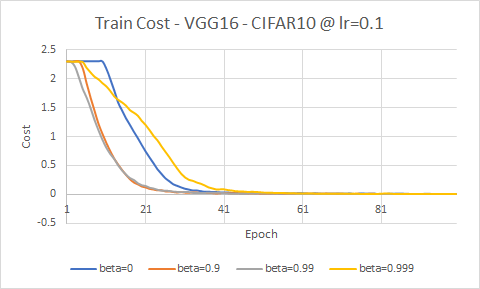


Figure . caption

after some experiments involving changing the value of , we find out the optimal value for in most cases is although maybe in some special cases tuning is required. while moving average helped gravity in initial speed but still we can see some delay. for solving this issue we propose an alternative value of . any specific value of averages over several previous data. we can find some data included in average by below [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 $t$ is small and , the value of equation 21 will be very small. there is a solution known as bias correction which modifies as follow:

(25)

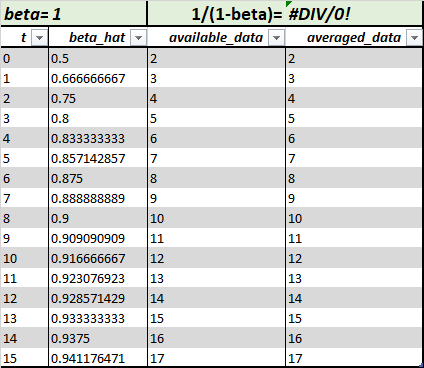
the logic behind equation 24 can be explained as follow: by increasing the value of the denominator approach to and equations 24 and 21will output almost the same result. but at the beginning of the training the value of is small and dividing equation 21 to this small value increases the total outcome.

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 equation 21 at initial steps with a different approach. we propose an alternative to beta as follow:

(26)

value of in equation 25 at large will tends to but at smaller values, it will correct the value of . let's drive equation 25. by choosing any value of we will average almost over data. let say we want to use a variable value of which increase over time and tends to , therefore always average over all of the data, we call this variable . for averaging over all of the data at each step we want the amount of data that we average to be (because at there is 2 data and ). then we can write:

at the value of is 0.5 which will average over 2 data ( and ). by increasing the value of the value of tends to which will average over all of the data. The below table shows the value of for different values of :



as you can see at each step the amount of data that we average on is equal to the total available data. for averaging over we modify the above equation as which is equation 25. now by increasing the value of the value of tends to . The below figure demonstrate behavior of for different values of .

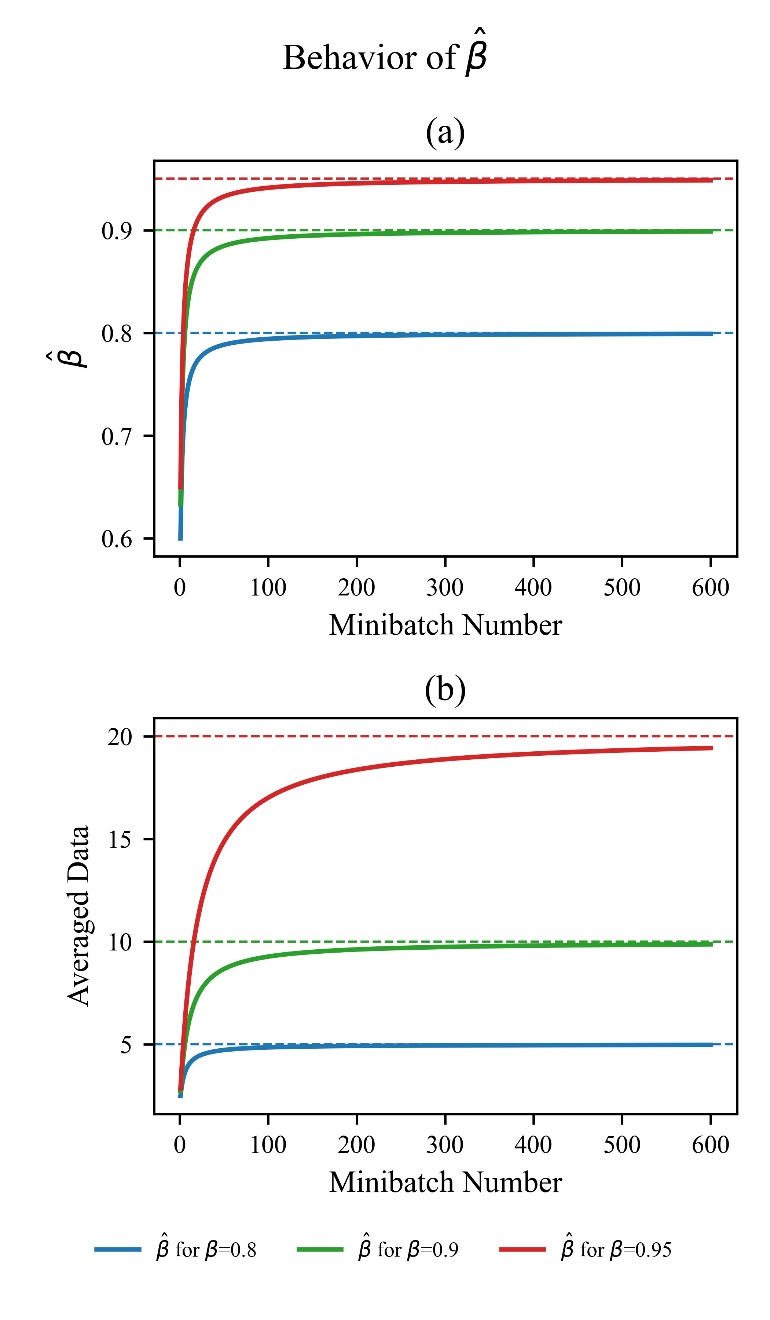


Figure caption

in addition to modifying in equation 21, we came out with another solution for increasing the speed of optimizer at early steps by using non-zero initial . instead of zero, we initialized with random numbers with normal distribution with mean as 0 and standard deviation as follow:

(27)

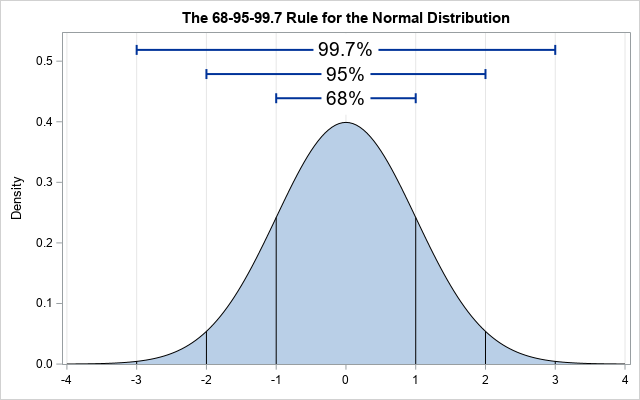
let’s look closely at the first update step:

(28)

we can think about this equation as two separated part first part is which is due to initial and second part which is due to the gradient term. we cannot do anything about the gradient term and this part is determined by many different parameters. but we can tweak for better initial loss reduction speed. first, let's define this term as follow:

(29)

by choosing we give total control of optimizer to an unknown part of equation 27 which is . for a normally distributed set of random numbers, 68% of random values are less than standard deviation and 95% of random number are less than and 99.7% of numbers are less than there for by choosing we define the range of initial steps for different parameters.



by choosing any 68% of parameters are . we define the numerator of the right side of this inequality as a new hyper-parameter called for controlling the value of initial steps:

(30)

equation 26 is the same different form of equation 29. by experimenting with different values of we found satisfying for the majority of models.

## 2.4 Summary

We propose a new optimizer for deep learning based on back-propagation from a kinematic point of view perspective. the summary of our 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 convenience and we prepared Keras implementation of gravity optimizer as follow:

# 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

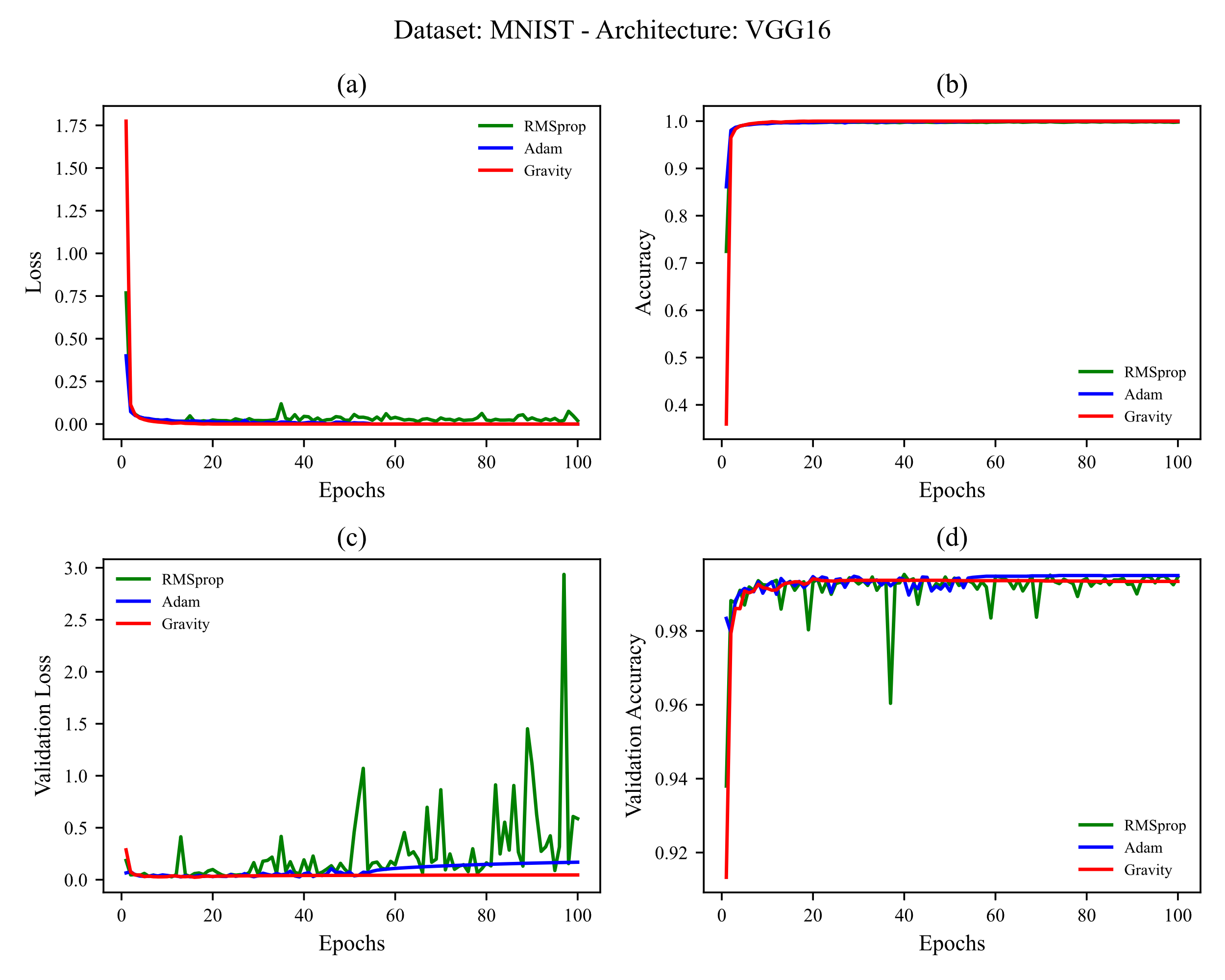


Figure 8. training MNIST on VGG16. learning rates used are shown in Table 4

### 4.1.2 MNIST Results on VGG19

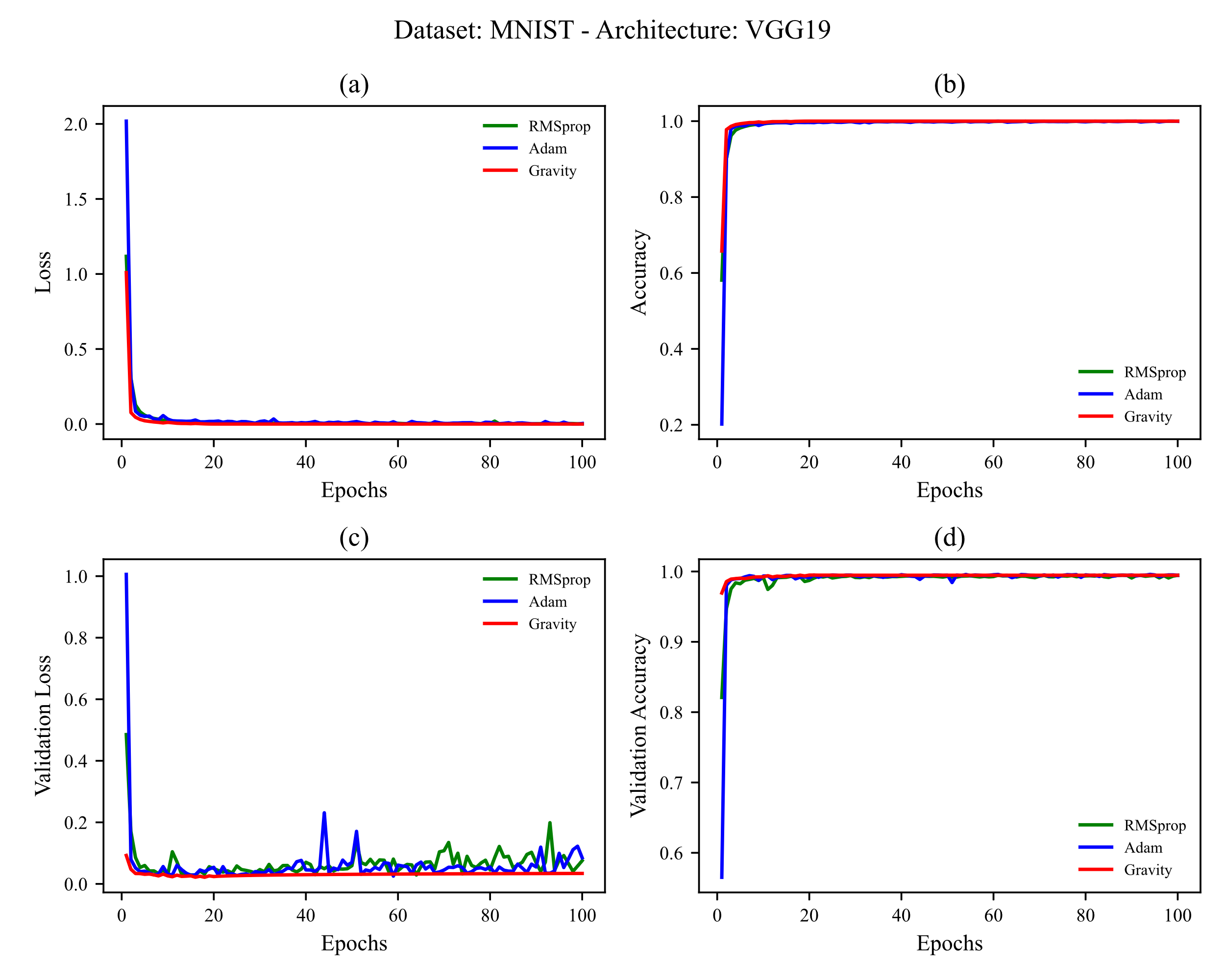


Figure 9 training MNIST on VGG19. learning rates used are shown in Table 4

## 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

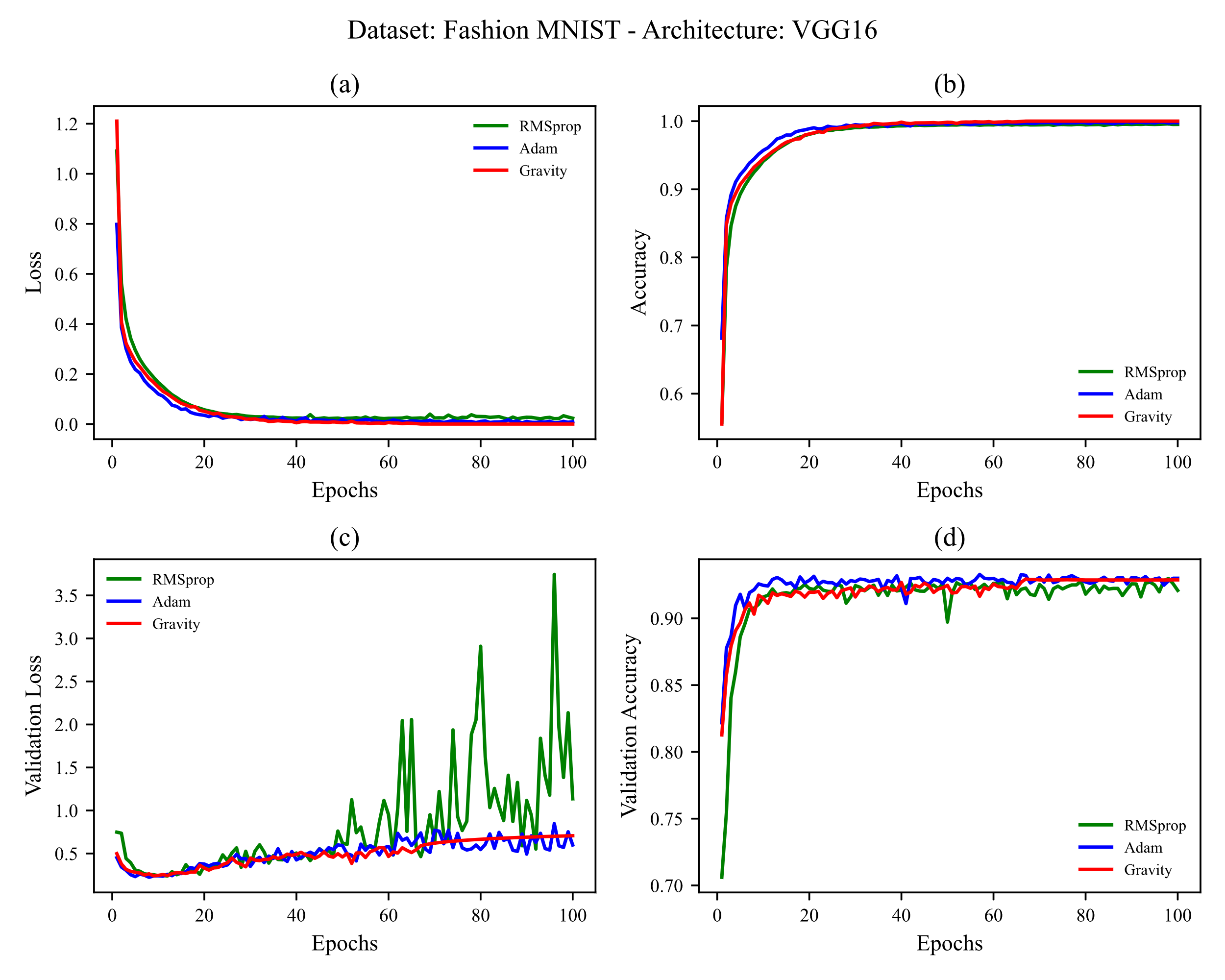


Figure 10 training Fashion-MNIST on VGG16. learning rates used are shown in Table 4

### 4.2.2 Fashion-MNIST on VGG19

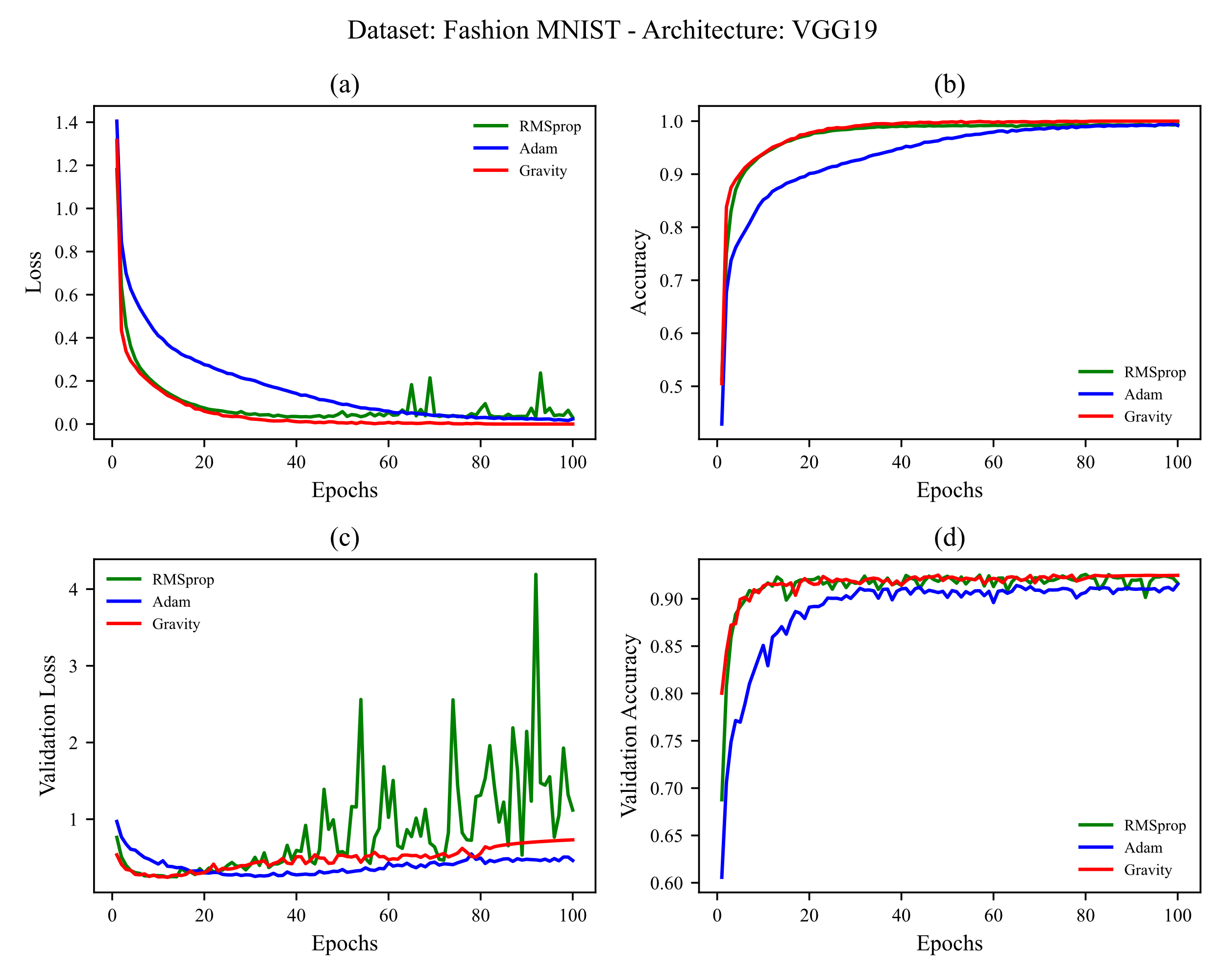


Figure 11 training Fashion-MNIST on VGG19. learning rates used are shown in Table 4

## 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

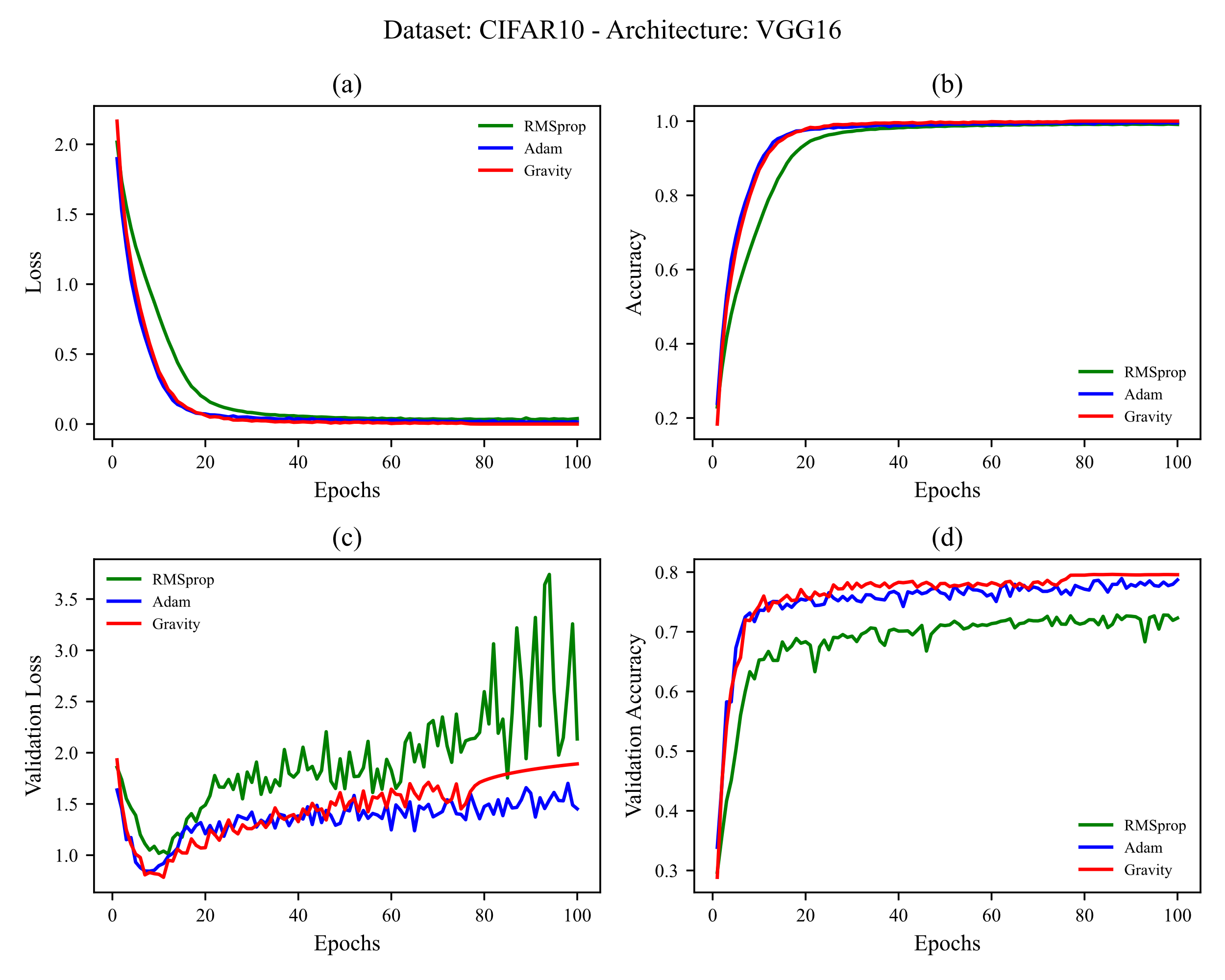


Figure 12 training CIFAR-10 on VGG16. learning rates used are shown in Table 4

### 4.3.1 CIFAR-10 on VGG19

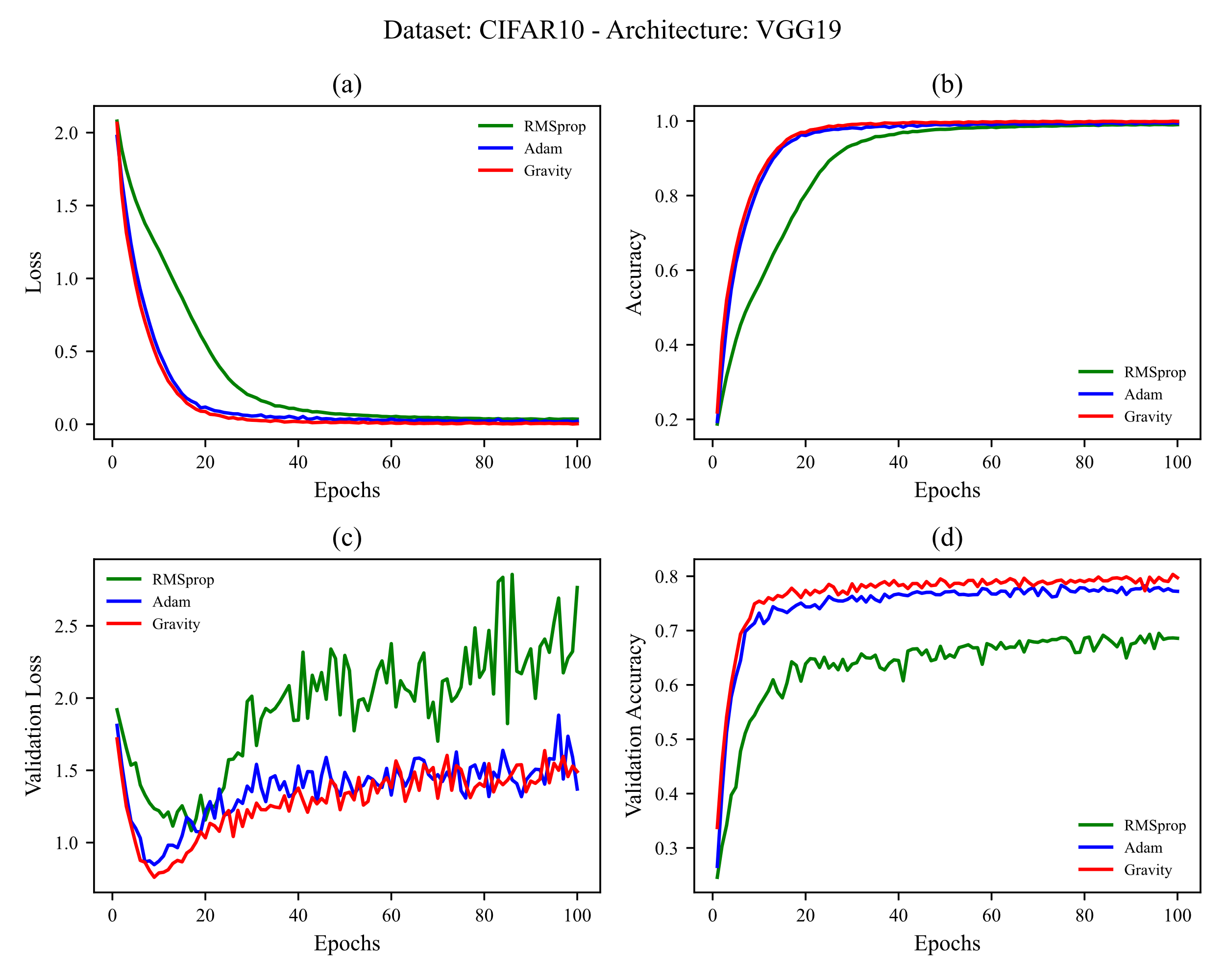


Figure 13 training CIDAR-10 on VGG19. learning rates used are shown in Table 4

## 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

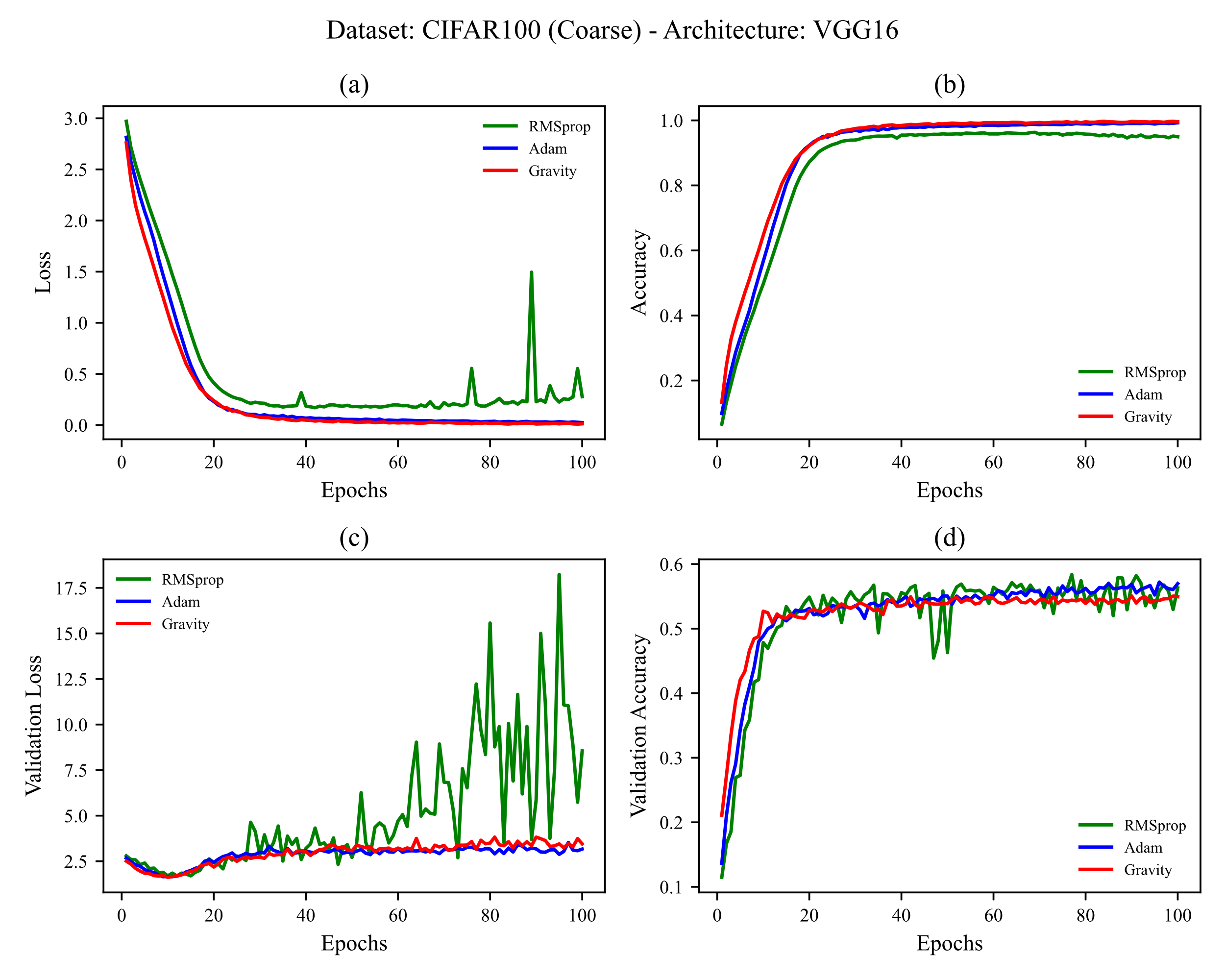


Figure 14 training CIFAR-100 (Coarse) on VGG16. learning rates used are shown in Table 4

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

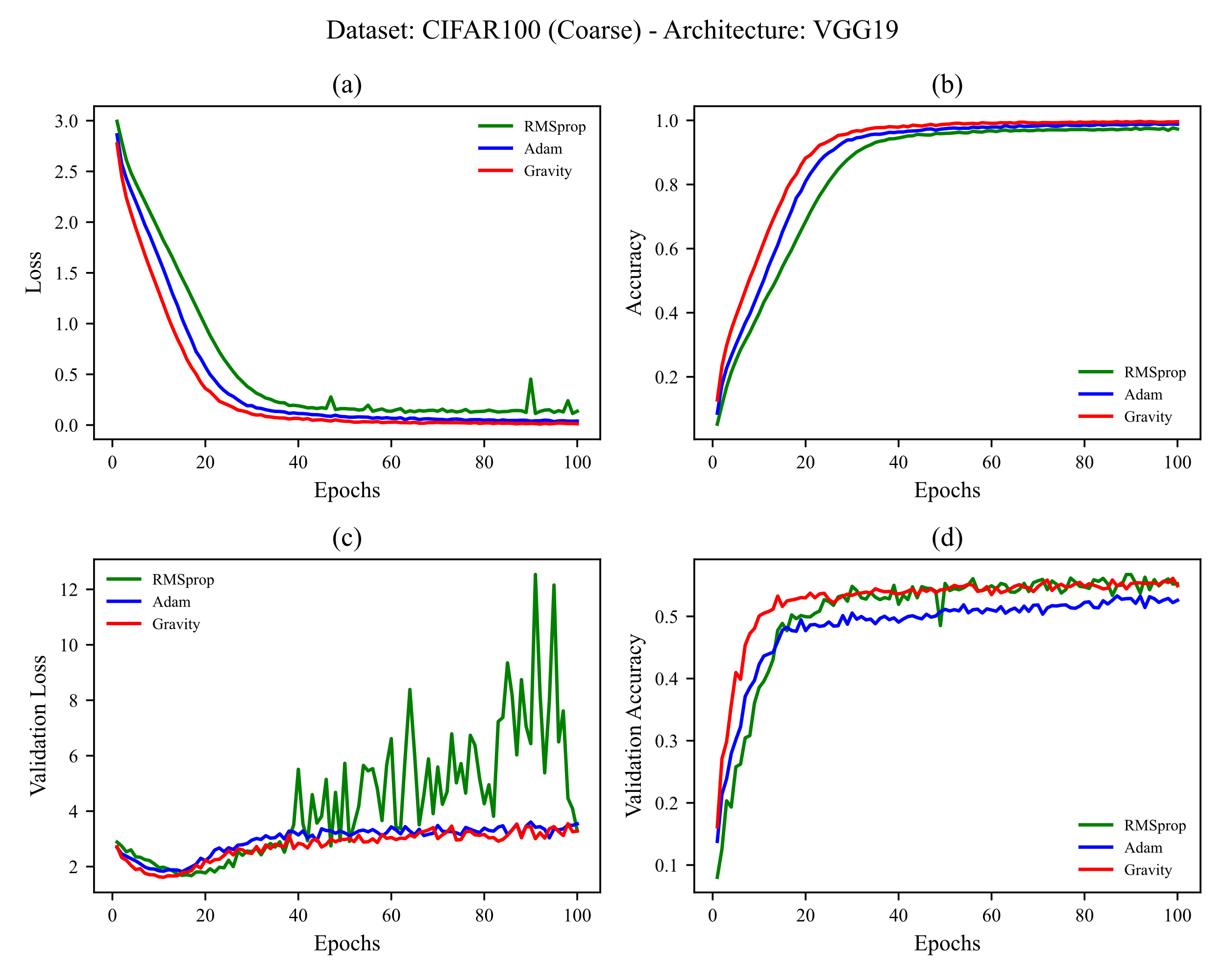


Figure 15 training CIFAR-100 (Coarse) on VGG19. learning rates used are shown in Table 4

## 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

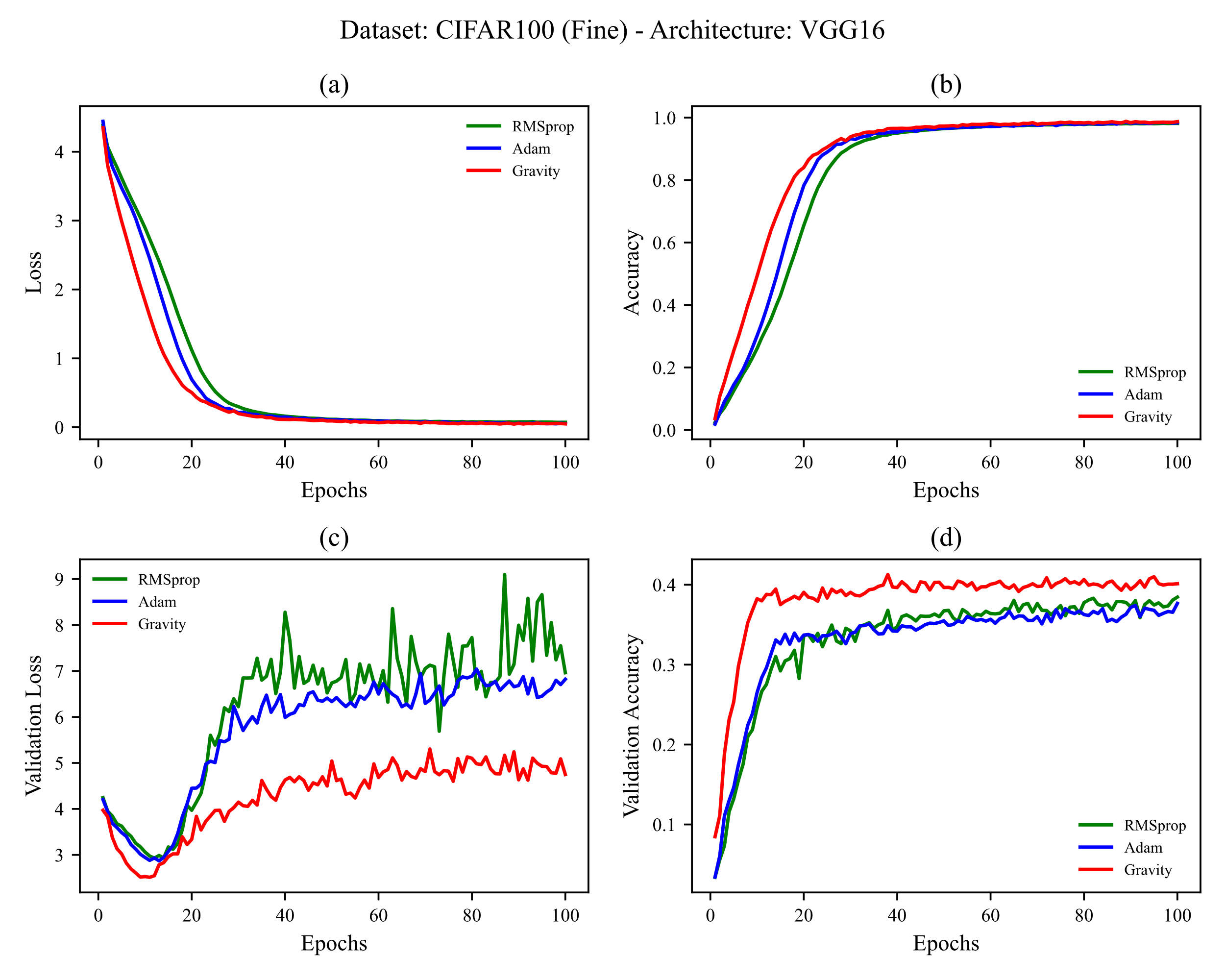


Figure 16 training CIFAR-100 (Fine) on VGG16. learning rates used are shown in Table 4

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

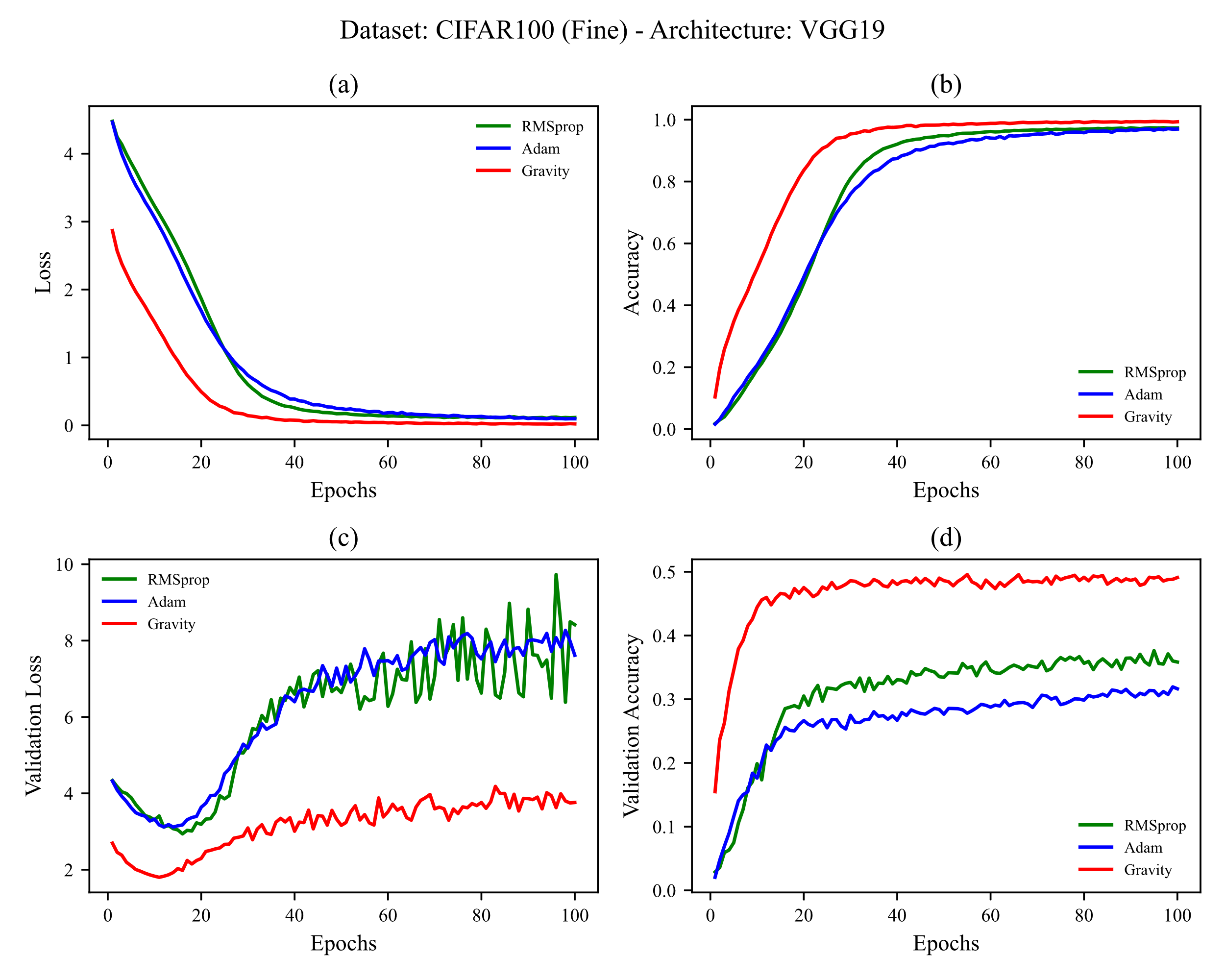


Figure 17 training CIFAR-100 (Fine) on VGG19. learning rates used are shown in Table 4

# 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.