**Gravity Optimizer: A Mechanical View on Optimization in Deep Learning**

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

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

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

**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. the most common standard optimizers in deep learning

|  |  |
| --- | --- |
| 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

# Version 2: بخش دوم میشه related works

**2 Algorithm**

**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 [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 super classes. 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 | 60 K | 10 K | 10 | 28x28x1 | 6 K |
| Fashion-MNIST | 60 K | 10 K | 10 | 28x28x1 | 6 K |
| CIFAR-10 | 50 K | 10 K | 10 | 32x32x3 | 5 K |
| CIFAR-100 (coarse) | 50 K | 10 K | 20 | 32x32x3 | 2.5 K |
| CIFAR-100 (fine) | 50 K | 10 K | 100 | 32x32x3 | 500 |

3.3 Architecture (models and hyperparameters)

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 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 VGGNets 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 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 hyperparameters 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 hyperparameters 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 trainings here are done with a batch size of 128 and for 100 epochs.

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 [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 Parametrs = 33,638,218** | | | **Total Parametrs = 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**

**3 Benchmark**

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 teach the same datasets we used. If results are not reported by other papers, the tests have been performed by us.

We used Google Colab (<colab.research.google.com>) as hardware because we couldn’t afford GPU for training deep neural network models on it and testing our idea. 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 this repository: <github.com/dariush-bahrami/gravity.optimizer>. On the other hand, with Google Colab and TensorFlow, we were given the chance to use TPUs (Tensor Processing Unit), Google’s custom-developed technology used to accelerate machine learning workloads[].

We used the following standard datasets to evaluate the performance of the 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 digits have been size-normalized and centered in a fixed-size image []. 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[]. 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 []. 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[]. Table 2 summarizes the detailed information of the datasets used.

Table 2. detailed information of datasets used in this paper for benchmark

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Dataset Name** | **#Train** | **#Test** | **#Class** | **Shape** | **Image per class** | **Ref** |
| **MNIST** | 60000 | 10000 | 10 | 28x28x1 | 6000 | [10] |
| **Fashion-MNIST** | 60000 | 10000 | 10 | 28x28x1 | 6000 |  |
| **CIFAR-10** | 50000 | 10000 | 10 | 32x32x3 | 5000 | [11] |
| **CIFAR-100 (coarse)** | 50000 | 10000 | 20 | 32x32x3 | 2500 |  |
| **CIFAR-100 (fine)** | 50000 | 10000 | 100 | 32x32x3 | 500 |  |

We used VGG16 and VGG19 with the exact specifications reported in their paper[]. VGG16 has about 34M and VGG19 has about 39M parameters[]. Although architectures such as ResNet50[] and EfficientNet[] have 23M and 4M parameters respectively, and they are as easy to implement as VGGNet in Keras, they showed so much slower training speed when implemented in Google Colab that (their model summaies for input shape of 32x32x3 and 10 classes are in [Gravity github repository](https://github.com/dariush-bahrami/gravity.optimizer)). Another reason why we have chosen VGG architecture over other architectures is that it doesn’t use any techniques to prevent overfitting. We want to minimize the impact of parameters such as Dropout[], Early Stopping[], L2 Regulation[], and Batch Normalization[] other than the optimizer on optimization. Finally, we monitor the loss and accuracy changes for the training and validation datasets for a constant number of epochs so that we can compare the performance of the Gravity optimizer to common standard optimizers listed in Table 1 with the same dataset and architecture without the impact of overfitting prevention techniques. Table 3 summarizes the models used in this paper.

Table 3. VGG model summaries used in the paper

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **VGG19** | | | | **VGG16** | | | |
| **Layer Type** | **Output Shape** | | **Parameters #** | **Layer Type** | **Output Shape** | | **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 |
| Conv2D | 8, 8, 256 | | 590,080 | MaxPooling2D | 4, 4, 256 | | 0 |
| MaxPooling2D | 4, 4, 256 | | 0 | Conv2D | 4, 4, 512 | | 1,180,160 |
| Conv2D | 4, 4, 512 | | 1,180,160 | Conv2D | 4, 4, 512 | | 2,359,808 |
| Conv2D | 4, 4, 512 | | 2,359,808 | 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 |
| MaxPooling2D | 2, 2, 512 | | 0 | Conv2D | 2, 2, 512 | | 2,359,808 |
| Conv2D | 2, 2, 512 | | 2,359,808 | 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 |
| MaxPooling2D | 1, 1, 512 | | 0 | Dense | 4096 | | 2,101,248 |
| **Dense Part** | | | | Dense | 4096 | | 16,781,312 |
| Flatten | 512 | | 0 | Dense | 10 | | 40,970 |
| Dense | 4096 | | 2,101,248 |  | | | |
| Dense | 4096 | | 16,781,312 |
| Dense | 10 | | 40,970 |
| **Total Parametrs:** | | **38,947,914** | | **Total Parametrs:** | | **33,638,218** | |

In Section 2, we talked about how to find the best values for the alpha, beta, and m hyperparameters in the gravity optimizer and why. Then, by performing various runs, we obtained and reported the best values. In this section, we used those suggested values for the benchmark. The remarkable thing about gravity optimizer is that in all the benchmarks we did (in all five datasets and two architectures used) the same values were considered for the hyperparameter and there was no need to change the hyperparameter to get a better result.

Our recommended value for learning rate, Alpha, and Beta is 0.1, 0.01, 0.9 respectively. We also set these values as default for Gravity optimizer in python implementation. In the next section, the results obtained from the training of five standard datasets with Gravity optimizers and two other standard and widely used optimizers (RMSProp and Adam) are compared. All trainings here are done with batch size of 128 and for 100 epochs.

3.1 MNIST

In this section, MNIST dataset was trained on VGG16 and VGG19 models and the results are compared with the results reported from other articles. More details of the results can be found in [github repository](https://github.com/dariush-bahrami/gravity.optimizer) or in materials section.

3.2 Fashion-MNIST

In this section, Fashion-MNIST dataset was trained on VGG16 and VGG19 models and the results are compared with the results reported from other articles. More details of the results can be found in [github repository](https://github.com/dariush-bahrami/gravity.optimizer) or in materials section.

3.3 CIFAR-10

In this section, CIFAR-10 dataset was trained on VGG16 and VGG19 models and the results are compared with the results reported from other articles. More details of the results can be found in [github repository](https://github.com/dariush-bahrami/gravity.optimizer) or in materials section.

3.4 CIFAR-100 (coarse)

In this section, CIFAR-100 (coarse) dataset was trained on VGG16 and VGG19 models and the results are compared with the results reported from other articles. More details of the results can be found in [github repository](https://github.com/dariush-bahrami/gravity.optimizer) or in materials section.

3.5 CIFAR-100 (fine)

In this section, CIFAR-100 (fine) dataset was trained on VGG16 and VGG19 models and the results are compared with the results reported from other articles. More details of the results can be found in [github repository](https://github.com/dariush-bahrami/gravity.optimizer) or in materials section.