Gradient Descent Optimization

Manthan Bharat Bhatt, Shubham Vipul Majmudar, Simran Singh

***Abstract*—Batch Gradient Descent is a first-order iterative optimization algorithm for finding the minimum of a function. It has been used in deep learning models to a great degree, whose objective is to find values of parameters that minimize a cost function. However, it suffers from a few limitations. For a large-enough data-set, it may not be possible to fit all the parameters and their gradients in memory while performing descent. Furthermore, given the number of parameters in a deep neural network and the complexity of loss functions, gradient descent may get stuck in local minima, saddle points or pathological curves. In this project, we describe variants of gradient descent—Mini-Batch Gradient Descent and Stochastic Gradient Descent, and also explain multiple optimization algorithms—Gradient Descent with Momentum, Nesterov's Accelerated Momentum, RMSProp and Adam Optimizer. We see how these techniques address the shortcomings of Batch Gradient Descent and also find the best hyper-parameters for each algorithm on the Fashion MNIST data-set and compare them with respect to their stability, learning speed and test data accuracy.**

***Index Terms*—Gradient Descent Optimization, Mini-Batch Gradient Descent, Polyak's Classical Momentum, Nestrov's Accelerated Momentum, RMSProp, Adam, Comparison, Analysis**

I. Introduction

With an increase in the availability of robust graphical processing units, achieving reasonable learning times for complex deep neural networks has become possible. There has been a rise in employing neural networks with hundreds of layers for learning models in various domains such as computer vision, speech recognition and natural language processing. As the size of a network increases, the number of parameters increase exponentially. AlexNet [1] that recorded a classification error of 15.3% on the ImageNet [2] database has 62.3 million parameters. ResNet-50, a 50 layered Residual Network [3], achieved 7.02% error over the same data-set with 25.6 million parameters. These parameters (weights and biases) are learnt by optimizing a loss function $L(w, b)$ through gradient descent. Some commonly used loss functions are the squared error loss function and the cross-entropy loss function. However, for such a high dimensional space, the loss function is never convex. Fig. 1 depicts contours of a VGG-56 [4] deep network's loss function on the CFAIR-10 [5] data-set. For such a loss contour, batch gradient descent faces certain limitations.

1) **Local Minima/Saddle Points**: Batch gradient descent can follow the direction of a local minima and get stuck. Saddle points are points on the contour where it is a global minima in one direction and local minima in other, and the contour is flatter towards the direction of local minima. Fig 2. shows an example of a saddle point in 3 dimensions.
2) **Pathological Curvature**: Pathological curves occur when the component of gradient in one direction is much larger than that in the component in the other direction. As shown in Fig 3., Pathological Curves are like ravines with gradually decreasing slopes.

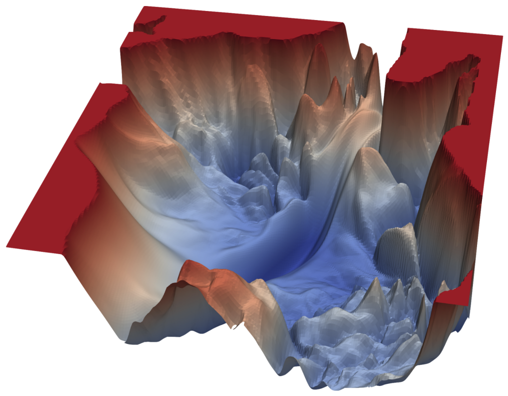


Fig. 1: 3D Contours of VGG-56 loss function [6]

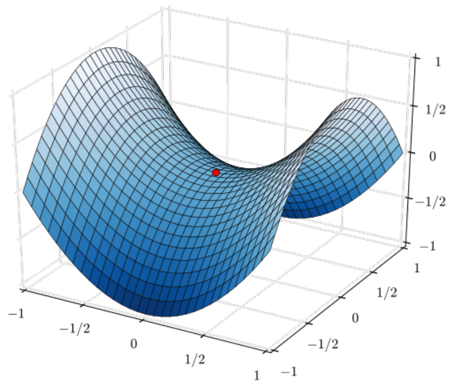


Fig. 2: Example of a Saddle Point

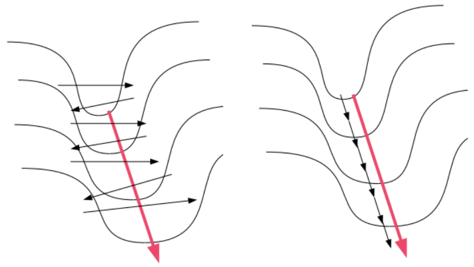


Fig. 3: Example of a Pathological Curve [7]

To overcome the first limitation, the most commonly used solution is Mini-Batch Gradient Descent or Stochastic Gradient Descent. Instead of using all training samples to calculate gradients, Mini-Batch Gradient Descent uses a small set (batch) of it. In this way, it introduces some amount of randomness is calculating the direction of gradients. It is possible that the direction of gradients for current batch is different than that of the entire data-set. Therefore, by taking that direction, gradient descent can escape local minima or saddle points. Stochastic Gradient Descent is Mini-Batch Gradient Descent with batch size of 1, i.e. just 1 sample is used to estimate gradients. To counter pathological curves, the information about degree of increase/decrease of gradients can be used by calculating second order derivatives. One such popular technique that uses second order derivatives is the Newton's method, which computes a matrix of the double derivatives of the loss function with respect of all combinations of the weights, called the Hessian Matrix. Note that if the second order derivative is positive, gradients are increasing with an increasing slope and vice-versa if the second order derivative is negative. But as mentioned before that the number of parameters in a deep neural network can be in millions (or even billions), and thus calculating the Hessian Matrix for all the parameters is not analytically tractable. However there are other methods that can estimate the direction and magnitude of the change of gradients and we explore them in this report.

1) **Gradient Descent with Momentum** [8] Gradient descent with momentum accumulates the gradient of the previous steps to determine the direction to descend. The moving average factor when multiplied aggressively steers the gradient towards the direction of maximum slope, while also damping the gradient where the gradient is of low magnitude, thus mitigating the problem of a steeping descent along certain direction. This can be thought of as analogous to a ball rolling down a hill. However, it may still not be able to move across efficiently through ravines along the pathological curvature. The following equations describe the exponentially weighted average

$$\begin{aligned} V_{dw} &= \gamma * V_{dw} + (1-\gamma) * dw \qquad (2)\\ V_{db} &= \gamma * V_{db} + (1-\gamma) * db \\ w &= w - \alpha * V_{dw} \\ b &= b - \alpha * V_{db} \end{aligned}$$

2) **RMSProp** [10] RMSProp, unofficially proposed by Dr Geoffrey Hinton, is especially helpful in resolving the problem of pathological curvature, since instead of multiplying the learning rate with a factor proportional to a power of past derivatives, it divides it with the decaying average of previous squared gradients to decay the learning rate. This is a form of gradient annealing in the direction where there is maximum descent, which is a good way to identify surrounding curvatures as the rate of gradient increases.

$$\begin{aligned} S_{dw} &= \beta * S_{dw} + (1-\beta) * dw^2 \qquad (3)\\ S_{db} &= \beta * S_{db} + (1-\beta) * db^2 \\ w &= w - \alpha * dw/\sqrt{S_{dw} + \epsilon} \\ b &= b - \alpha * db/\sqrt{S_{db} + \epsilon} \end{aligned}$$

3) **Adam** [11] Adam combines the heuristics of RMSProp and gradient descent with momentum. Hence, we calculate the exponential average as well as the squares of the gradient of each parameter.

$$\begin{aligned} V_{dw}^{corrected} &= V_{dw}/(1-\gamma^t) \\ V_{db}^{corrected} &= V_{db}/(1-\gamma^t) \\ S_{dw}^{corrected} &= S_{dw}/(1-\beta^t) \\ S_{db}^{corrected} &= S_{db}/(1-\beta^t) \\ w &= w - \alpha * V_{dw}^{corrected}/\sqrt{S_{dw}^{corrected} + \epsilon} \\ b &= b - \alpha * V_{db}^{corrected}/\sqrt{S_{db}^{corrected} + \epsilon} \end{aligned}$$

4) **Nesterov's Accelerated Momentum** [9] The problem with momentum is that once we get close to the local minima, the momentum is very high and it does not know that it should slow down. This could result in missing the local minima point and diverging. Hence in the algorithm, the gradient is not computed for the current weights, but for the current weights added with the beta times the previous value of exponentially weighted average term. The convergence rate is proportional to $1/n$ where n is the number of iterations. Nesterov's algorithm improves the convergence rate tremendously by making it proportional to $1/n^2$. Considering the ball analogy, the basic difference between momentum and Nesterov's method is the order of jumps made by the ball. In momentum, we first compute the gradient, then make a jump in that direction and finally make an update. In Nesterov's algorithm, we first make a jump based on the previous momentum, then calculate the gradient and finally make an update. This anticipatory move prevents in going too fast or too slow. The beauty of Nesterov's algorithm is that the updates are adapted according to the slopes of the gradient.

$$\begin{aligned} V_{dw} &= \beta * V_{dw} + (1-\beta) * d(w + (\beta * V_{dw})) \\ V_{db} &= \beta * V_{db} + (1-\beta) * d(b + (\beta * V_{db})) \\ w &= w - \alpha * V_{dw} \\ b &= b - \alpha * V_{db} \end{aligned}$$

II. Related Work

In this section, we discuss some already presented works that aim to achieve the same goal. [13] discusses the alternative forms of gradient descent algorithms and concludes mini-batch performs the best. The paper also elaborates numerous algorithms for optimizing the cost function such as Momentum, Nesterov Accelerated Gradient, RMSProp, Adam, AdaMax and Nadam. Furthermore, the paper also proposes techniques to improve Stochastic Gradient Descent namely batch normalization, early stopping and curriculum learning.

The problem with saddle points and local minima is that they often have plateaus of smaller curvature in their neighborhood. The works of [15] tackle this problem by proposing an algorithm for second-order optimization problem which rapidly avoids the saddle points in high dimension to compute the gradient.

[14] in their work discuss about the problem of convergence of a function to a local minima. The paper states that majority of non-convex optimization problems have 2 characteristics: 1) all local minima are also global minima, and 2) near any saddle point, the objective function has a negative directional curvature. These characteristics imply that gradient can further continue to descend and thus the global minima can be found out with the use of normal gradient descent algorithm.

III. Method

For comparing different gradient descent techniques and optimization algorithms, we train a neural network model that learns to classify images from the Fashion MNIST [12] data-set. The network has 3 fully connected layers with 500, 100 and 10 neurons in each layer respectively. As each image in the Fashion MNIST [12] data-set is of 28x28 pixels, the input layer of our neural network has 784 neurons. To compare and analyze all four gradient optimization methods, we take the following approach:

1) For finding the ideal learning rate for each algorithm, we run each algorithm for 5 different learning rates - $0.1, 0.03, 0.01, 0.003$ and 0.001 and document the training costs and test accuracies. We also plot their training cost vs iterations graphs to see how fast the said algorithm converges for different values of learning rates. Note that all these experiments are run with a batch-size of 512 and coefficient of first moment γ and coefficient of second moment β set to their optimal values of 0.9 and 0.999 respectively.
2) Once we have found the ideal parameters for each algorithm, we compare them with respect to their speed of convergence, stability and test-data accuracy achieved.
3) At last, we compare the performance of Mini-Batch Gradient Descent with different batch sizes on Adam [11]. We try with batch size 1 (Stochastic Gradient Descent), $64, 512$ and 1024 and note their stability as a function of number of iterations.

IV. Experiments

In this section, we list all the experiments conducted and the results that we obtained.

A. Tuning Learning Rates

For each optimization algorithm, we train our network multiple times while varying learning rate. We have fixed the batch-size to 512 and coefficient of first moment γ and coefficient of second moment β are set to their optimal values of 0.9 and 0.999 respectively. Following are the results that we obtained:

1) **Classical Gradient Descent** The graph of training cost and number of iterations can be seen in Fig. 4. Table 1 documents final training cost, test accuracy and number of training epochs taken to converge, for each value of learning rate.

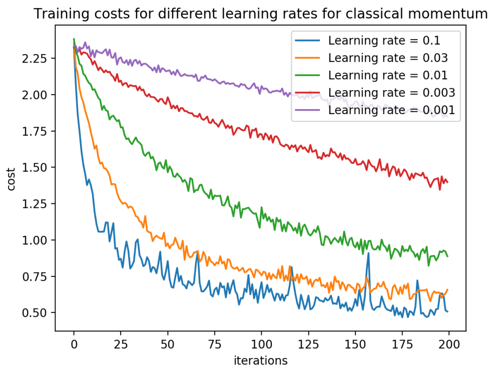


Fig. 4: Training Costs of Classical Gradient Descent vs Number of Iterations

TABLE I: Classical Gradient Descent Results

Learning Rate	Cost	Accuracy(%)	Epochs
0.1	0.507	81.94	4
0.03	0.657	75.32	10
0.01	0.888	71.76	13
0.003	1.398	63.44	18
0.001	1.849	50.62	24

2) **Gradient Descent with Momentum** Fig. 5 depicts the convergence of Gradient Descent with Momentum with different learning rates. Training cost, test accuracy and the number of epochs required to converge are listed in Table 2.

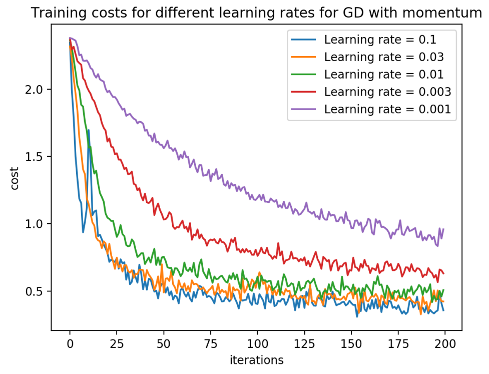


Fig. 5: Training Costs of Gradient Descent with Momentum vs Number of Iterations

TABLE II: Gradient Descent with Momentum Results

Learning Rate	Cost	Accuracy(%)	Epochs
0.1	0.356	85.38	1
0.03	0.422	84.10	1
0.01	0.507	82.54	1
0.003	0.631	78.16	2
0.001	0.958	71.52	5

3) **Nesterov's Accelerated Momentum** The graph in Fig. 6 demonstrates convergence of training cost with Nesterov's Accelerated Momentum. For test accuracy and

number of epochs taken to converge, refer to Table 3.

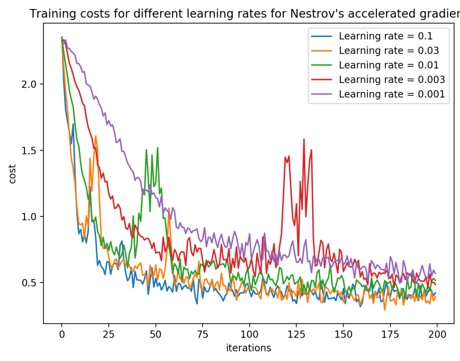


Fig. 6: Training Costs of Gradient Descent with Nesterov's Accelerated Momentum vs Number of Iterations

TABLE III: Nesterov's Accelerated Momentum Results

Learning Rate	Cost	Accuracy(%)	Epochs
0.1	0.511	82.20	1
0.03	0.387	83.90	1
0.01	0.486	82.98	1
0.003	0.531	80.90	3
0.001	0.577	78.92	2

4) **RMSProp** Fig. 7 depicts the convergence of RMSProp with different learning rates. Training cost, test accuracy and the number of epochs required to converge are listed in Table 4.

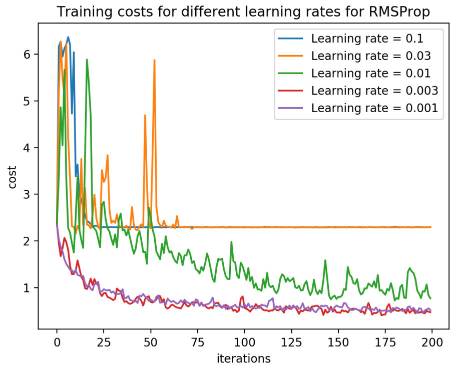


Fig. 7: Training Costs of RMS-Prop vs Number of Iterations

TABLE IV: RMSProp Results

Learning Rate	Cost	Accuracy(%)	Epochs
0.1	2.300	10.00	1
0.03	2.295	10.00	1
0.01	0.773	63.54	2
0.003	0.475	83.30	1
0.001	0.519	79.96	1

5) **Adam** The graph of training cost and number of iterations can be seen in Fig. 4. Table 1 documents final training cost, test accuracy and number of training epochs taken to converge, for each value of learning rate.

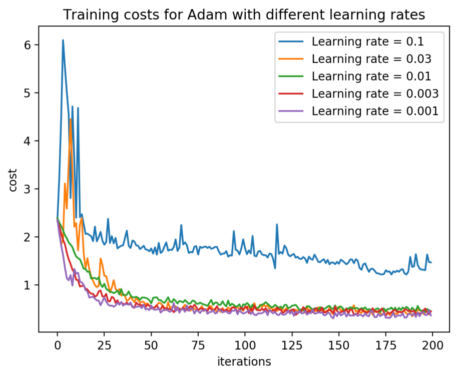


Fig. 8: Training Costs of Adam vs Number of Iterations

TABLE V: Adam Results

Learning Rate	Cost	Accuracy(%)	Epochs
0.1	1.564	34.3	2
0.03	0.409	84.3	1
0.01	0.551	82.52	1
0.003	0.479	83.46	1
0.001	0.356	85.10	1

B. Comparison

Now that we have tuned the hyper-parameters of each algorithms, we make a comparative analysis between them and document our results.

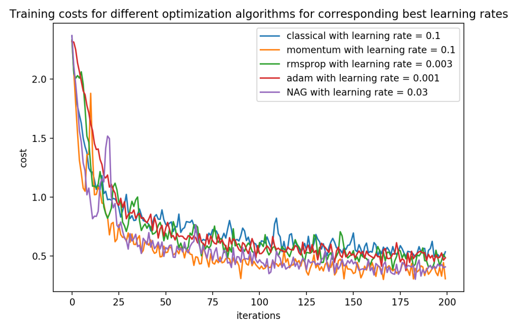


Fig. 9: Training Cost vs Iterations for all optimization algorithms

C. Mini-Batch Gradient Descent

To observe the effect of varying batch-size, we train our network with batch sizes $1, 64, 256$ and 1024. Fig 10 depicts the decrease in training costs with iterations for each batch-size.

V. Conclusion

From the study performed on the Fashion-MNIST [12] dataset, we obtained insights on the performance of various

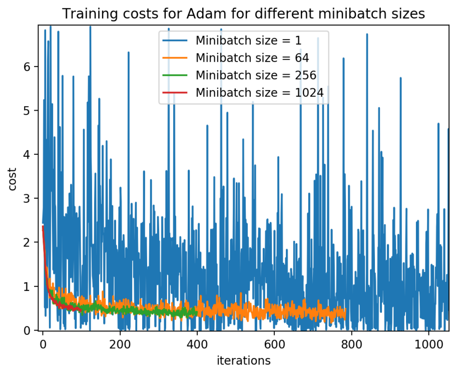


Fig. 10: Training Cost vs Iterations for different mini-batch sizes using Adam Optimizer

gradient descent optimization algorithms, and were able to compare it with the intuition that led to their inception.

After running our experiments and observing results, we can infer the following points:

1) Gradient Descent takes longer than any other algorithm to converge (as can be seen from the number of epochs in Table 1).
2) Gradient descent with momentum is an improved version of the classical gradient descent, and takes shorter number of iterations to converge than the classical gradient descent. Furthermore, it is a more stable algorithms and given better test accuracy (Table 2).
3) Nesterov's Accelerated Gradient performs better than classical gradient descent and its test accuracy and convergence time are similar to that of gradient descent with Polyak's momentum. However, it is much more unstable and diverges with small changes in learning rate. Also, highest oscillations of gradients are observed in Nesterov's Accelerated Gradient.
4) Both RMSProp and Adam have similar convergence time and stability. However, Adam resulted in slightly better test-set accuracy.
5) From Fig. 10 we can infer that as we increase the batch size, our optimization algorithm becomes stable. This is because we have more number of samples to estimate gradient from and therefore, the estimate of the gradient will be near to the actual gradient.

VI. Division of Work

Following is how the work in this project was divided between team members:

- **Manthan Bharat Bhatt**: Studied Adam and Nestrov's accelerated gradient. Also laid out much of the content for the report.
- **Shubham Vipul Majmudar**: Studied on gradient descent with momentum and RMS Prop, and analyzed and plotted the outputs, and assissted in the creation of the report.
- **Simran Singh**: Researched on RMSProp and Adam, and studied related literature-survey papers on the same; helped in the completion of report.

VII. Self-Peer Evaluation Table

TABLE VI: Evaluation

Team Member	Rating
Manthan Bharat Bhatt	20
Shubham Vipul Majmudar	20
Simran Singh	20

References

[1] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, in: *Advances in Neural Information Processing Systems 25 (NIPS 2012).*

[2] ImageNet Dataset, *http://www.image-net.org/*

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun, *Deep Residual Learning for Image Recognition*, in: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*

[4] Karen Simonyan, Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, in: *arXiv preprint arXiv:1409.1556*

[5] CFAIR-10 Dataset, *https://www.cs.toronto.edu/~kriz/cifar.html*

[6] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer and Tom Goldstein, *Visualizing the Loss Landscape of Neural Nets*, in:*ICLR 2018 Conference*

[7] James Martens, *Deep learning via Hessian-free optimization*, in: *Proceedings of the 27 th International Conference on Machine Learning, Haifa, Israel, 2010*

[8] B. T. Polyak, *Some methods of speeding up the convergence of iteration methods*, in: *USSR Computational Mathematics and Mathematical Physics, vol. 4, no. 5, pp. 117, 1964.*

[9] Y. E. Nesterov, *A method for solving the convex programming problem with convergence rate o (1/k 2)* in: *Dokl. Akad. Nauk SSSR, vol. 269, 1983, pp. 543547.*

[10] T. Tieleman and G. Hinton, *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude* in: *COURSERA: Neural networks for machine learning, vol. 4, no. 2, pp. 2631, 2012*

[11] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization* in: *arXiv preprint arXiv:1412.6980, 2014.*

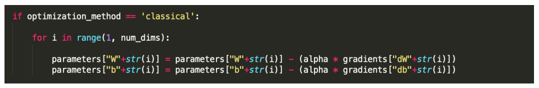
[12] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. (2017)*

[13] Sebastian Ruder, *An overview of Gradient Descent Optimization algorithms* in: *arXiv:1609.04747*

[14] Ju Sun, Qing Qu and John Wright, *When Are Nonconvex Problems Not Scary?* in: *arXiv:1510.06096*

[15] Dauphin Y., Pascanu R., Gulcehre C., Cho K., Ganguli S. and Bengio Y., *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization* in: *arXiv:1406.2572v1*

Code Snippets

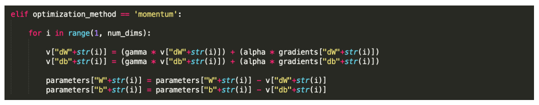


```
if optimization_method == 'classical':

    for i in range(1, num_dims):

        parameters["W"+str(i)] = parameters["W"+str(i)] - (alpha * gradients["dW"+str(i)])
        parameters["b"+str(i)] = parameters["b"+str(i)] - (alpha * gradients["db"+str(i)])
```

Fig. 11: Classical



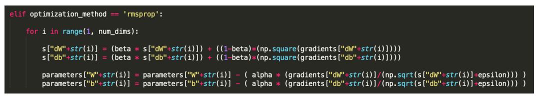
```
elif optimization_method == 'momentum':

    for i in range(1, num_dims):

        v["dW"+str(i)] = (gamma * v["dW"+str(i)]) + (alpha * gradients["dW"+str(i)])
        v["db"+str(i)] = (gamma * v["db"+str(i)]) + (alpha * gradients["db"+str(i)])

        parameters["W"+str(i)] = parameters["W"+str(i)] - v["dW"+str(i)]
        parameters["b"+str(i)] = parameters["b"+str(i)] - v["db"+str(i)]
```

Fig. 12: GD with momentum



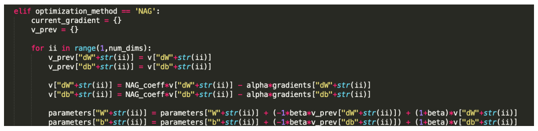
```
elif optimization_method == 'rmsprop':

    for i in range(1, num_dims):

        s["dW"+str(i)] = (beta * s["dW"+str(i)]) + ((1-beta)*(np.square(gradients["dW"+str(i)])))
        s["db"+str(i)] = (beta * s["db"+str(i)]) + ((1-beta)*(np.square(gradients["db"+str(i)])))

        parameters["W"+str(i)] = parameters["W"+str(i)] - ( alpha * (gradients["dW"+str(i)]/(np.sqrt(s["dW"+str(i)]+epsilon))) )
        parameters["b"+str(i)] = parameters["b"+str(i)] - ( alpha * (gradients["db"+str(i)]/(np.sqrt(s["db"+str(i)]+epsilon))) )
```

Fig. 13: RMSProp



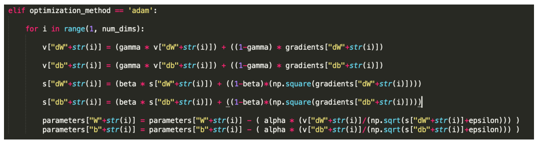
```python
elif optimization_method == 'NAG':
    current_gradient = {}
    v_prev = {}

    for ii in range(1,num_dims):
        v_prev["dW"+str(ii)] = v["dW"+str(ii)]
        v_prev["db"+str(ii)] = v["db"+str(ii)]

        v["dW"+str(ii)] = NAG_coeff*v["dW"+str(ii)] - alpha*gradients["dW"+str(ii)]
        v["db"+str(ii)] = NAG_coeff*v["db"+str(ii)] - alpha*gradients["db"+str(ii)]

        parameters["W"+str(ii)] = parameters["W"+str(ii)] + (-1*beta*v_prev["dW"+str(ii)]) + (1+beta)*v["dW"+str(ii)]
        parameters["b"+str(ii)] = parameters["b"+str(ii)] + (-1*beta*v_prev["db"+str(ii)]) + (1+beta)*v["db"+str(ii)]
```

Fig. 14: Nestrov’s accelerated gradient



```python
elif optimization_method == 'adam':

    for i in range(1, num_dims):

        v["dW"+str(i)] = (gamma * v["dW"+str(i)]) + ((1-gamma) * gradients["dW"+str(i)])

        v["db"+str(i)] = (gamma * v["db"+str(i)]) + ((1-gamma) * gradients["db"+str(i)])

        s["dW"+str(i)] = (beta * s["dW"+str(i)]) + ((1-beta)*(np.square(gradients["dW"+str(i)])))

        s["db"+str(i)] = (beta * s["db"+str(i)]) + ((1-beta)*(np.square(gradients["db"+str(i)])))

        parameters["W"+str(i)] = parameters["W"+str(i)] - ( alpha * (v["dW"+str(i)]/(np.sqrt(s["dW"+str(i)]+epsilon))) )
        parameters["b"+str(i)] = parameters["b"+str(i)] - ( alpha * (v["db"+str(i)]/(np.sqrt(s["db"+str(i)]+epsilon))) )
```

Fig. 15: Adam