

Compare different SG-Schemes based on large least square problems

Ramkrishna Acharya

December 16, 2024

Abstract

This study reviews some of the popular stochastic gradient-based schemes based on large least-square problems. These schemes, often called optimizers in machine learning play a crucial role in finding better parameters of a model. Hence this study focuses on viewing such optimizers with different hyper-parameters and analyzing them based on least square problems.

1 Introduction

1.1 Least Squares Problems

Least squares problems are common problems in which we try to find the best-fitting curve to a given set of points by minimizing the sum of the squares of the offsets ("the residuals") of the points from the curve. Alternatively, they can be defined as a parameter estimation method in regression analysis based on minimizing the sum of squared residuals (RSS).

Let:

- $\mathbf{y} \in \mathbb{R}^n$ be the vector of observed values
- $\mathbf{X} \in \mathbb{R}^{n \times p}$ be the matrix of input features
- $\boldsymbol{\theta} \in \mathbb{R}^p$ be the vector of regression coefficients (or parameters)

Then such a least squares problem can be defined as the matrix form of:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\epsilon} \quad (1)$$

where $\boldsymbol{\epsilon} \in \mathbb{R}^n$ represents the error terms, typically assumed to be normally distributed.

Then the sum of squared residuals (RSS) can be written as:

$$\text{RSS} = (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) \quad (2)$$

Alternatively, in terms of the error vector $\boldsymbol{\epsilon}$, RSS can be expressed as:

$$\text{RSS} = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon}$$

$$\text{RSS} = \boldsymbol{\epsilon}^T \boldsymbol{\epsilon} := (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) \quad (3)$$

The parameters $\boldsymbol{\theta}$ can be found by minimizing the sum of squared residuals. Expanding on the above expression:

$$\boldsymbol{\epsilon}^T \boldsymbol{\epsilon} = \mathbf{y}^T \mathbf{y} - 2\boldsymbol{\theta}^T \mathbf{X}^T \mathbf{y} + \boldsymbol{\theta}^T \mathbf{X}^T \mathbf{X} \boldsymbol{\theta}$$

To minimize RSS, we take the derivative with respect to $\boldsymbol{\theta}$ and set it to zero:

$$\frac{\partial \boldsymbol{\epsilon}^T \boldsymbol{\epsilon}}{\partial \boldsymbol{\theta}} = -2\mathbf{X}^T \mathbf{y} + 2\mathbf{X}^T \mathbf{X} \boldsymbol{\theta} = 0$$

Solving for $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (4)$$

where $\mathbf{X}^T \mathbf{X}$ is assumed to be invertible.

¹Eric W. Weisstein. *Least Squares Fitting*. MathWorld—A Wolfram Web Resource. Accessed: July 1, 2024. 2024. URL: <https://mathworld.wolfram.com/LeastSquaresFitting.html>

2 Methodology

2.1 Least Squares and Gradient Scheme

Gradient descent is an optimization algorithm used to minimize some convex functions by iteratively moving in the direction of the steepest descent as defined by the negative of the gradient. The parameter update technique is given by 4 i.e. $\hat{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$ could have problems:

- $(\mathbf{X}^T \mathbf{X})^{-1}$ might not be invertible.
- Our \mathbf{X} might be too big and cause storage problems while loading it.
- The number of observations (rows in \mathbf{X}) could be smaller than features (columns in \mathbf{X}).

But, we could find the best parameters $\hat{\theta}$ with an iterative update method. Let's define $\mathbf{f}(\mathbf{X}; \theta) = \mathbf{X}\theta + \epsilon$ as a regression function, and $J(\mathbf{f}(\mathbf{X}; \theta), \mathbf{y})$ be a convex loss function we try to minimize. Then an example of such an iterative method can be shown in the algorithm 1.

Algorithm 1 Simple Gradient Descent Algorithm for Least Squares Regression

```
1: Initialize parameters  $\theta$ , learning rate (step length)  $\eta$ 
2: while stopping criterion not met do
3:   for each data point  $(\mathbf{X}^{(i)}, y^{(i)})$  in  $(\mathbf{X}, \mathbf{y})$  do
4:     Compute gradient estimate:  $g = \frac{\partial}{\partial \theta} J(f(X^{(i)}; \theta), y^{(i)})$ 
5:     Update parameters:  $\theta \leftarrow \theta - \eta g$ 
6:   end for
7: end while
```

2.2 Different Choices to Make While Updating Parameters

In machine learning, we use gradient descent to update the parameters of our model. In our case, we try to fit a linear regression model and try to find the best parameters θ . The following section explains some of the different choices one can make:

- Loss Functions : Example: Mean Squared Error (MSE), Mean Absolute Error (MAE), etc.
- Batch Size : Example: Batch size of 1, batch size of data counts, and any real values in between.
- Optimizers : Example: Variable learning rate, constant learning rate, momentum, etc.

2.2.1 Loss Functions

Loss functions are the convex functions we try to minimize. By finding the gradients with respect to the parameters, we find the direction and magnitude needed for our parameter to be updated. Hence it plays a crucial role in gradient schemes. Mean Squared Error and Mean Absolute Error are most common examples of convex functions.

1. Mean Squared Error (MSE)

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N (y^{(i)} - f(X^{(i)}; \theta))^2 := \frac{1}{N} \sum_{i=1}^N j_i(\theta)$$

where $y^{(i)}$ are the observed values, and N is the number of data points. MSE is smooth and convex in nature and a good choice for regression tasks.

2. Mean Absolute Error (MAE)

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N |y^{(i)} - f(X^{(i)}; \theta)| := \frac{1}{N} \sum_{i=1}^N j_i(\theta)$$

In MSE, we need to calculate the squared term but here in MAE, we calculate the absolute of difference between target and predicted values. It is easier to compute in the sense that we do not need to calculate squared now but it is not a smooth function. Furthermore, it is not differentiable at $y^{(i)} = f(X^{(i)}; \theta)$.

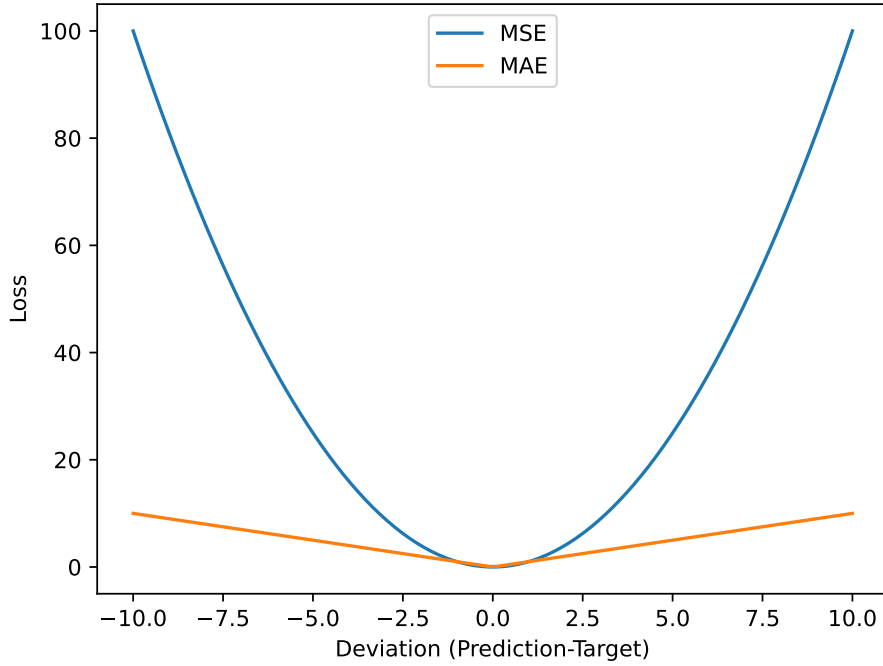


Figure 1: MSE vs MAE

2.2.2 Gradient Descent Variants based on Batch Size

1. Batch Gradient Descent

Batch gradient descent, also known as vanilla gradient descent, computes the gradient of the loss function for the parameters θ for the entire training dataset. The following algorithm 2 shows how it performs parameter updates.

Algorithm 2 Batch Gradient Descent Algorithm

- 1: Initialize parameters θ , learning rate (step length) η
 - 2: **while** stopping criterion not met **do**
 - 3: Compute gradient estimate: $g = \frac{1}{N} \sum_{i=1}^N \frac{\partial j_i(\theta)}{\partial \theta}$
 - 4: Update parameters: $\theta \leftarrow \theta - \eta g$
 - 5: **end while**
-

As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. Batch gradient descent also doesn't allow us to update our model online, i.e. with new examples on the fly. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and a local minimum for non-convex surfaces. [5]

2. Stochastic Gradient Descent

Stochastic gradient descent (SGD) performs a parameter update for each training example $(x^{(i)}, y^{(i)})$. The following algorithm 3 shows how it performs parameter updates.

Algorithm 3 Stochastic Gradient Descent Algorithm

- 1: Initialize parameters θ , learning rate (step length) η
 - 2: **while** stopping criterion not met **do**
 - 3: Randomly choose single data point $(\mathbf{X}^{(i)}, y^{(i)})$ from (\mathbf{X}, \mathbf{y})
 - 4: Compute gradient estimate: $g = \frac{\partial}{\partial \theta} j_i(\theta)$
 - 5: Update parameters: $\theta \leftarrow \theta - \eta g$
 - 6: **end while**
-

While batch gradient descent performs redundant computations for large datasets with similar examples, SGD does parameter updates one at a time. Hence it can be used to learn online.

3. Mini-Batch Gradient Descent Mini-batch gradient descent takes the best of both batch and stochastic gradient descent and performs a parameter update for every mini-batch of B training examples. The following algorithm 4 shows how it performs parameter updates.

Algorithm 4 Mini-Batch Gradient Descent Algorithm

- 1: Initialize parameters θ , learning rate (step length) η , and B batch size.
 - 2: **while** stopping criterion not met **do**
 - 3: Randomly sample B examples.
 - 4: Compute gradient estimate: $g = \frac{1}{B} \sum_{b=1}^B \frac{\partial}{\partial \theta} j_b(\theta)$
 - 5: Update parameters: $\theta \leftarrow \theta - \eta g$
 - 6: **end while**
-

Remarks On Gradient Descent Variants Based on Batch Size

- As we are updating parameters after calculating only one gradient in stochastic gradient descent, it will be faster and need little memory to hold gradients, but updating too frequently causes gradients to be noisy and computationally inefficient. But this can also help escape the local minimum and find the global one.
- Using batch gradient descent, we update parameters based on gradients of all data points so loading large datasets is problematic. While it could generate smoother losses, calculating the gradients for all data points is time-consuming.
- Using mini-batch gradient descent, we update parameters based on a small batch of the data points which will be smoother than stochastic gradient descent and faster than batch gradient descent.

It is also worth noting that if we want to perform parameter updates for 1000 iterations, using a batch size of 1, the gradient calculation would be done for only 1000 samples, using a batch size of 32, the gradient calculation would be done for $32 * 1000$ samples, and using a batch size of 900, the gradient calculation would be done for $900 * 1000$ samples.

2.2.3 Gradient Descent Variants based on Parameter Update [5]

These techniques often called optimizers in machine learning come in different variations each with their pros and cons. The simplest parameter update techniques are already given in the previous sections in algorithm 1. Hence we can start by pointing out its pros and cons. Even though it is easier to calculate and faster, simple gradient descent often oscillates around the local minima. Furthermore, it updates parameters without the knowledge of its past gradient information. To overcome this nature, a momentum optimizer [4] was found.

1. Momentum Optimizer

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. To achieve this, this algorithm adds a fraction γ of the previous timestep's update value to the current update value.

Let $g_{t,k}$ be the gradient of a parameter k with respect to Loss $J(\theta)$ at time step t , then we can write it as follows:

$$g_{t,k} = \nabla_{\theta_k} J(\theta_t)$$

The momentum term for the current timestep of parameter k is calculated as:

$$v_{t,k} = \gamma v_{t-1,k} + \eta g_{t,k} \tag{5}$$

Then the parameter update rule becomes:

$$\theta_{t+1,k} = \theta_{t,k} - v_{t,k} \tag{6}$$

Where,

- v_{t-1} is the update vector of the previous time step and is initialized as a zero vector at $t = 0$.
- γ is a momentum coefficient ranging from 0 to 1 and is usually set to 0.9.
- When γ is near 0, it becomes similar to gradient descent but when it is near 1, helps in faster convergence and reduces oscillations.

2. Nesterov accelerated gradient

In momentum optimizer, we move our parameters based on the momentum term without caring about where it is heading towards. Now Nesterov Accelerated Gradient (NAG) calculates the gradient not w.r.t. to our current parameters θ_t but w.r.t. the approximate future position of our parameters θ_t . For a k^{th} parameter at time step t , we perform parameter update as follows:

$$\begin{aligned} v_{t,k} &= \gamma v_{t-1,k} + \eta \nabla_{\theta_k} J(\theta_t - \gamma v_{t-1}) \\ \theta_{t+1,k} &= \theta_{t,k} - v_{t,k} \end{aligned} \quad (7)$$

Here γ , a momentum term is set around 0.9 again.



Figure 2: Nesterov update (Source: G. Hinton's lecture 6c)

While Momentum first computes the current gradient (small blue vector in the above Figure) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector), NAG first makes a big jump in the direction of the previously accumulated gradient (brown vector), measures the gradient and then makes a correction (green vector).

3. Adaptive Gradient Method (Adagrad) [1]

Let $g_{t,k}$ be the gradient of a parameter k with respect to Loss $J(\theta)$ at time step t , then we can write it as follows:

$$g_{t,k} = \nabla_{\theta_k} J(\theta_t)$$

In previous methods, we had a constant learning rate for all parameters. However, this is problematic for frequent and infrequent parameters. Adagrad adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.

Let G_t be the vector to contain the sum of squares of gradients and can be written for parameter k as:

$$G_{t,k} = G_{t-1,k} + g_{t,k}^2$$

Then the update rule is:

$$\theta_{t+1,k} = \theta_{t,k} - \frac{\eta}{\sqrt{G_{t,k} + \varepsilon}} \cdot g_{t,k} \quad (8)$$

Here, ε is a smoothing term that avoids division by zero (usually on the order of 10^{-8}). G_t at $t = 1$ is a zero vector. Now we have an adaptive learning rate, but as we accumulate squared gradients in the denominator, the adapted learning rate can shrink quickly.

4. RMSprop [7]

Instead of the sum of squares of the gradient, we now use the running average of the squares of the gradients. For parameter θ_k at time step t , we can define the running average of the square of the gradients as:

$$E[g^2]_{t,k} = \gamma E[g^2]_{t-1,k} + (1 - \gamma) g_{t,k}^2$$

The parameter update rule is:

$$\theta_{t+1,k} = \theta_{t,k} - \frac{\eta}{\sqrt{E[g^2]_{t,k} + \epsilon}} g_{t,k} := \theta_{t,k} - \frac{\eta}{\text{RMS}[g]_{t,k}} g_{t,k}$$

Where RMS is the root mean squared criterion of the gradient. Hence the name **RMS**. The author (G. Hinton) suggested using $\gamma = 0.9$ and $\eta = 0.001$.

5. ADADELTA [9]

The authors[9], mention that this optimizer was derived to improve the two main drawbacks of Adagrad, 1) the continual decay of learning rates throughout training, and 2) the need for a manually selected global learning rate.

Although being developed independently around the same time, RMSProp and Adadelata work similarly around the RMS term. Authors first derived the update rule as:

$$\theta_{t+1,k} = \theta_{t,k} - \frac{\eta}{RMS[g]_{t,k}} g_{t,k}$$

But authors [9] note that units in these steps do not match (i.e. if the parameters had some hypothetical units, the changes to the parameter should be changes in those units as well) so they defined **running average of squared parameter updates**:

$$E[\Delta\theta^2]_{t,k} = \gamma E[\Delta\theta^2]_{t-1,k} + (1 - \gamma) \Delta\theta_{t,k}^2$$

And its RMS is,

$$RMS[\Delta\theta]_{t,k} = \sqrt{E[\Delta\theta^2]_{t,k} + \epsilon}$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous timestep to get a new parameter update rule:

$$\theta_{t+1,k} = \theta_{t,k} - \eta \frac{RMS[\Delta\theta]_{t-1,k}}{RMS[g]_{t,k}} g_{t,k} \quad (9)$$

The original implementation does not have any learning rate i.e. $\eta = 1$.

6. ADAM (Adaptive Moment Estimation) [3]

Adam combines RMSProp's exponentially decaying average of past squared gradients (v_t) and exponentially decaying average of past gradients (like momentum, m_t). For parameter θ_k at time t :

$$\begin{aligned} m_{t,k} &= \beta_1 m_{t-1,k} + (1 - \beta_1) g_{t,k} \\ v_{t,k} &= \beta_2 v_{t-1,k} + (1 - \beta_2) g_{t,k}^2 \end{aligned}$$

Here, β_1 and β_2 are called decay rates.

As m_t (first moment of gradients) and v_t (second moment of gradients) are initialized as vectors of zeros, they tend to bias towards 0 in early timesteps when decay rates are small. Hence authors proposed bias-corrected moment estimates.

$$\begin{aligned} \hat{m}_{t,k} &= \frac{m_{t,k}}{1 - \beta_1^t} \\ \hat{v}_{t,k} &= \frac{v_{t,k}}{1 - \beta_2^t} \end{aligned}$$

Then the parameter update rule becomes:

$$\theta_{t+1,k} = \theta_{t,k} - \frac{\eta}{\sqrt{\hat{v}_{t,k}} + \epsilon} \hat{m}_{t,k} \quad (10)$$

The authors propose default values of 0.9 for β_1 , 0.999 for β_2 , and 10^{-8} for ϵ .

3 Experiments and Results

In this section, above mentioned optimizers are experimented on the toy dataset of the least squares problem. The dataset is prepared pseudo-randomly and has the following properties:

- Random Seed: 100
- Input Data: $\mathbf{X} \in \mathbb{R}^{1000 \times 5}$
- Random data $\mathbf{X} \sim \mathcal{U}(0, 100)$
- Normalize data by dividing by max value: $\mathbf{X}_{\text{norm}} = \frac{\mathbf{X}}{\max(\mathbf{X})}$
- Parameters (or Weights): $\boldsymbol{\theta} \sim \mathcal{N}(0, 1)$, $\boldsymbol{\theta} \in \mathbb{R}^5$
- Bias: $b \sim \mathcal{N}(0, 1)$, $b \in \mathbb{R}$
- Noise: $\epsilon \sim \mathcal{N}(0, 1)$, $\epsilon \in \mathbb{R}^{1000}$
- Output: $\mathbf{y} = \mathbf{X}_{\text{norm}}\boldsymbol{\theta} + b + 0.1\epsilon$, $\mathbf{y} \in \mathbb{R}^{1000}$

Our goal will be to find the parameters θ which will be able to minimize the loss function.

To evaluate the performance of the parameter in each epoch, we randomly split X_{norm} into 90 : 10 ratios. The first half will be used to calculate loss and gradients to update parameters while the second half will be used to evaluate the parameter's performance on unseen data. The first half is often called **training data** and the second half is called **validation data**.

3.1 Experiments Setup

For this experiment, we can have:

- variations of loss function as 1 i.e. MSE
- variations of batch size: 1, 32, whole dataset
- variations of learning rates (if applicable): 0.1, 0.01, 0.001, and for momentum: 0.1, 0.9
- variations of optimizers are 7 (SGD, Momentum, Nesterov, Adagrad, RMSProp, Adadelta, Adam)
- number of epochs (i.e. number of iterations) as 1000.

Experiments are done with the following settings:

- PyTorch is used for training a linear model. Because it handles gradient calculations and provides optimizer's implementations.
- Parameters in the linear model are initialized $\mathcal{U}(-\frac{1}{\sqrt{5}}, \frac{1}{\sqrt{5}})$

Note: PyTorch implementation of Momentum and NAG optimizers are slightly different than the proposed by ¹. But the PyTorch’s implementation has been used in some popular works like DenseNet ². In previous slides, we showed implementation by ³.

	[6] Implementation	PyTorch Implementation
Momentum	$v_t = \mu v_{t-1} - \eta \nabla_{\theta} J(\theta_{t-1})$ $\theta_t = \theta_{t-1} + v_t$	$v_t = \mu v_{t-1} + \nabla_{\theta} J(\theta_{t-1})$ $\theta_t = \theta_{t-1} - \eta v_t$
NAG	$v_t = \mu v_{t-1} - \eta \nabla_{\theta} J(\theta_{t-1} + \mu v_{t-1})$ $\theta_t = \theta_{t-1} + v_t$	$v_t = \mu v_{t-1} + \nabla_{\theta} J(\theta_{t-1})$ $g_t = g_t + \mu v_t$ $\theta_t = \theta_{t-1} - \eta g_t$

Table 1: Comparison of Original and PyTorch Implementations of Momentum and NAG

Codes that produced results in this work are available on [GitHub](#).

3.2 Results

Based on the learning rates, we can group optimizers into two groups i.e. constant learning rate optimizers and adaptive learning rate optimizers.

3.2.1 Validation Loss On Constant LR Optimizers

We have SGD, Momentum, and Nesterov adaptive gradient for the constant learning rate-based optimizers.

1. Stochastic Gradient Descent Optimizer

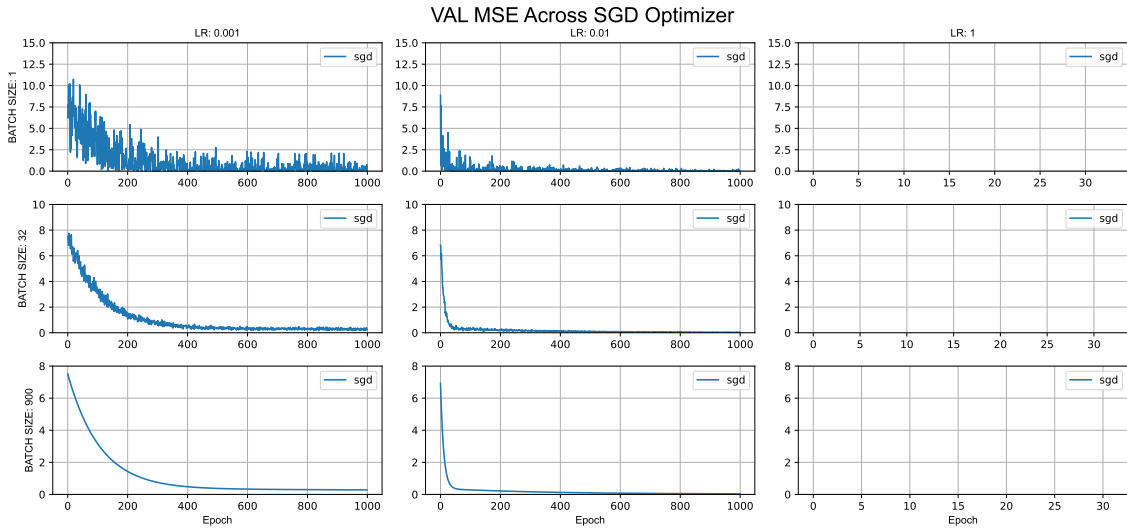


Figure 3: Validation MSE While using SGD

From figure 3 we can observe that:

- Using a batch size of 1 (i.e. stochastic gradient descent), the loss decreases with too much noise i.e. unstable. It is an expected behavior when we update parameters based on the gradient of only one training example. By the end of 1000 iterations, we calculated gradients for only 1000 samples.
- Using a batch size of 32 (i.e. mini-batch gradient descent), the loss decreases a little smoother than the batch size of 1. It is because, in each epoch, we are updating parameters based on the gradients of 32 training examples. By the end of 1000 iterations, we calculated gradients for 32*1000 samples.

¹Ilya Sutskever, James Martens, and Geoffrey E Hinton. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*. 2013

²Gao Huang et al. *Densely Connected Convolutional Networks*. 2018. arXiv: 1608.06993 [cs.CV]. URL: <https://arxiv.org/abs/1608.06993>

³Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG]

- Using a batch size of 900 (i.e. batch gradient descent), loss values are much smoother and do not oscillate either. It is because in each epoch we are updating parameters based on the gradients of 900 training examples. By the end of 1000 iterations, we calculated gradients for 900*1000 samples.
- As the learning rate increases, we are experiencing less loss quickly but at learning rate 1, everything diverges. It happened because the gradients were too large and hence parameters were i.e. gradient explosion.

2. Momentum Optimizer

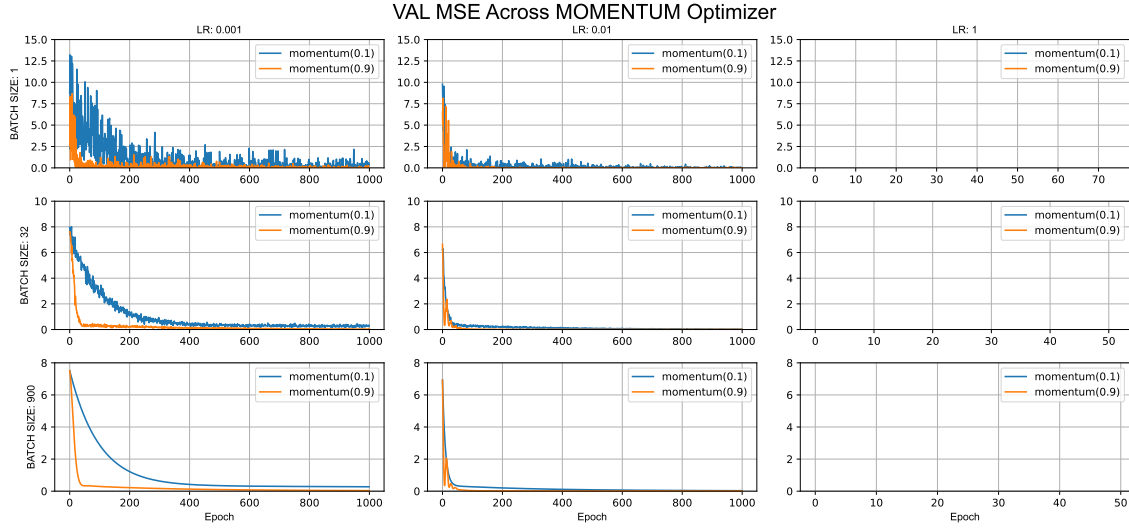


Figure 4: Validation MSE While using Momentum Optimizer

From figure 4 we can observe that:

- Loss curves follow a pattern like when using SGD. i.e. as batch size increases loss is less noisy and at learning rate 1, the loss was too high and training was aborted.
- When the momentum rate is too low, it behaves more like SGD, but when it is high we see more stable and smaller losses.

3. Nesterov Optimizer

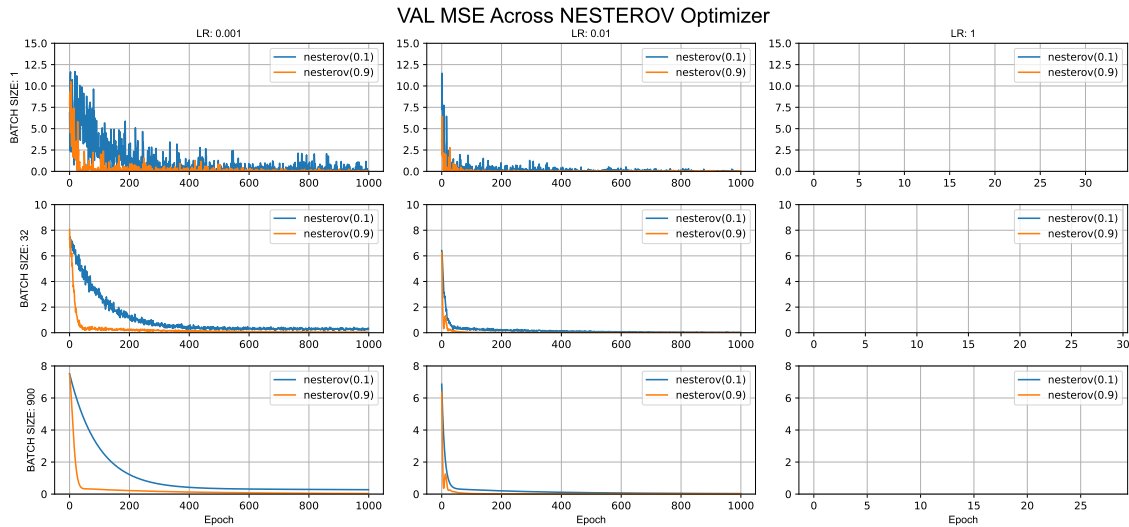


Figure 5: Validation MSE While using Nesterov Optimizer

From figure 5 we can observe that:

- Loss curves follow a pattern like a momentum optimizer. i.e. as batch size, momentum, and rate increases loss is less noisy, and at learning rate 1, the loss was too high and training was aborted.

Finally, we can compare them in a single plot.

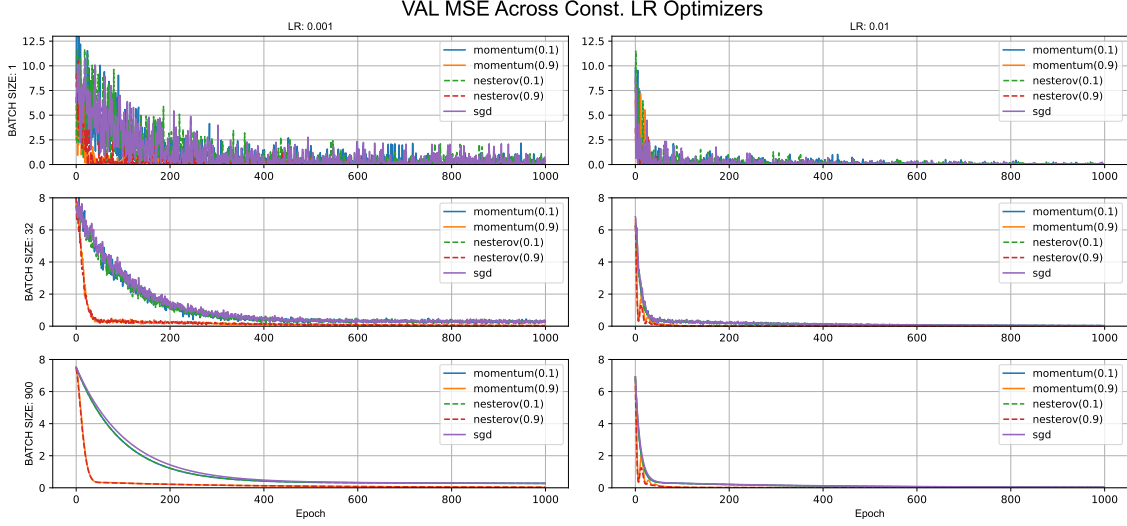


Figure 6: Validation MSE across constant LR optimizers

From figure 6 we can observe that:

- Nesterov and Momentum both have almost similar loss curves for this data.
- Using Nesterov, we can see slightly smoother loss curves than momentum with a batch size of 32 and batch size of 900.

3.2.2 Validation Loss On Adaptive LR Optimizers

We have Adagrad, RMSProp, Adadelta, and Adam for the adaptive learning rate-based optimizers.

1. Adagrad Optimizer

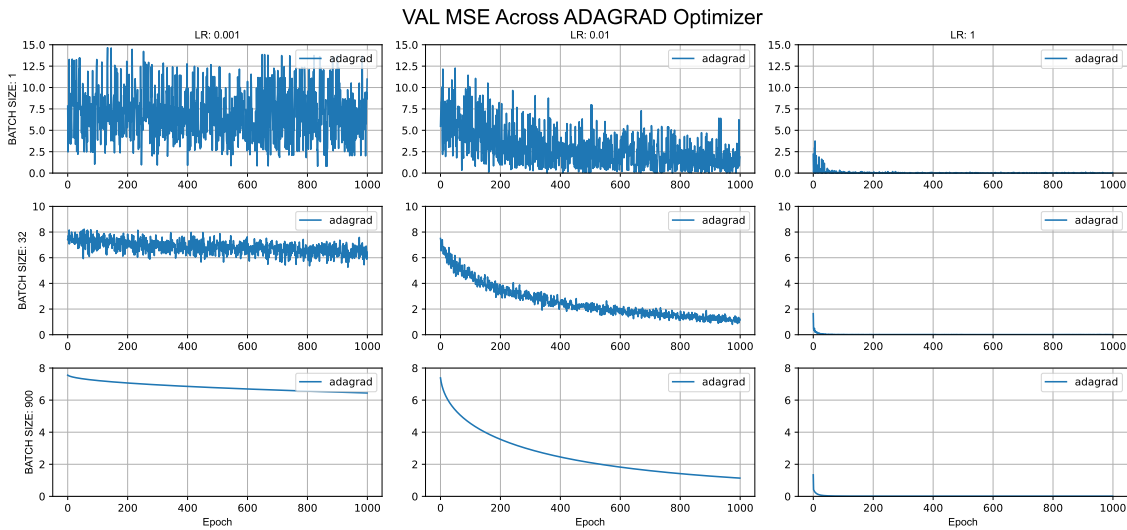


Figure 7: Validation MSE While using Adagrad Optimizer

From figure 7 we can observe that:

- It seems higher learning rates have reduced loss more and now gradient explosion also did not happen at a learning rate of 1. It is because we do not have a fixed learning rate now but it is being adapted and different for different parameters.

2. RMSProp Optimizer

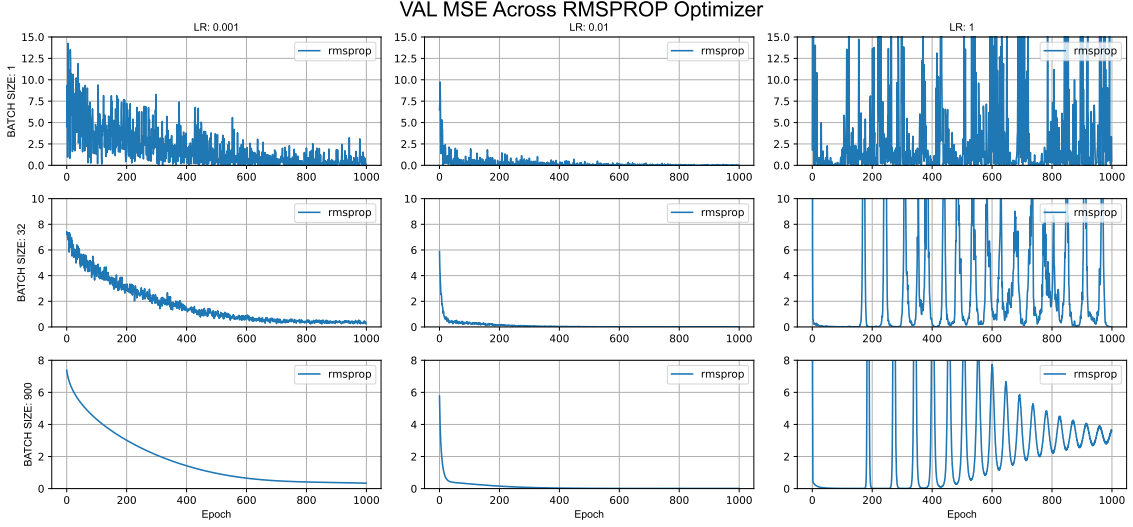


Figure 8: Validation MSE While using RMSProp Optimizer

From figure 8 we can observe that:

- A learning rate of 1 gives giving very small loss in the early epoch but it shows very unstable loss behavior.

3. Adadelta Optimizer

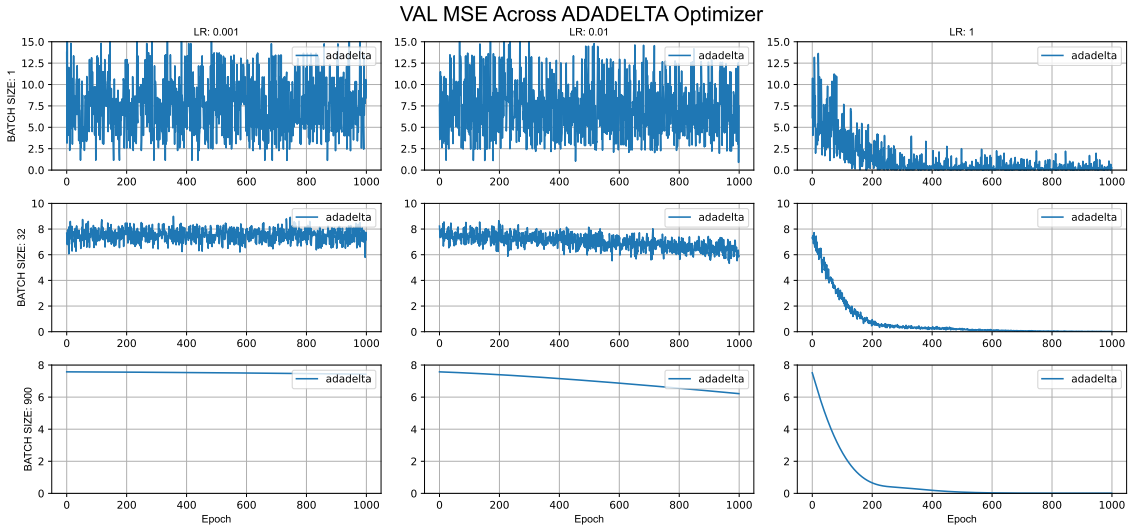


Figure 9: Validation MSE While using Adadelta Optimizer

From figure 9 we can observe that:

- A learning rate of 1 is giving smoother and better results than others. This is expected because authors [9] used a learning rate of 1.

4. Adam Optimizer

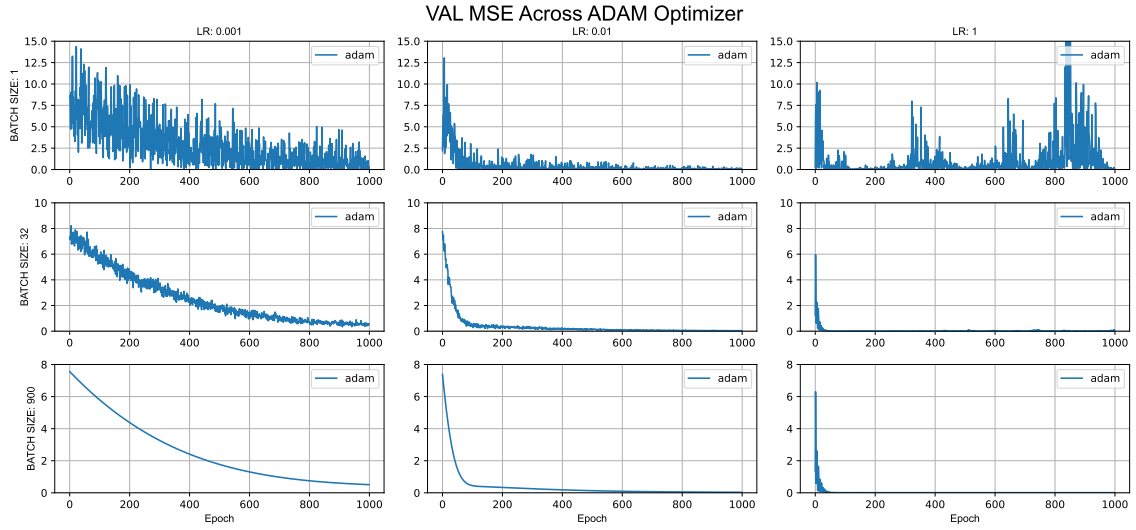


Figure 10: Validation MSE While using Adam Optimizer

From figure 10 we can observe that:

- At a learning rate of 1 and batch size of 1, Adam is unstable but increasing batch size shows stable behavior.

Now we can look at all these 4 results in a single plot to make comparisons.

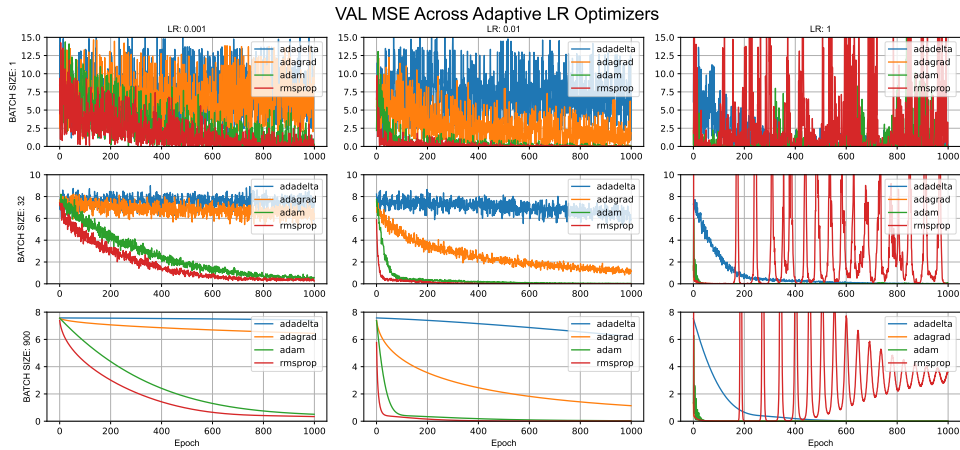


Figure 11: Validation MSE across adaptive LR optimizers

From figure 11 we can observe that,

- Only when the learning rate is 1, is the loss reduced faster for Adadelta.
- Other than RMSProp, optimizers quickly reached lesser loss values at a learning rate of 1.
- Adam and RMSProp quickly achieved a smaller validation MSE than others but RMSProp shows instability at a learning rate of 1.

We can look into the Nesterov and Adam optimizers for comparison as well because these two are the better-performing optimizers.

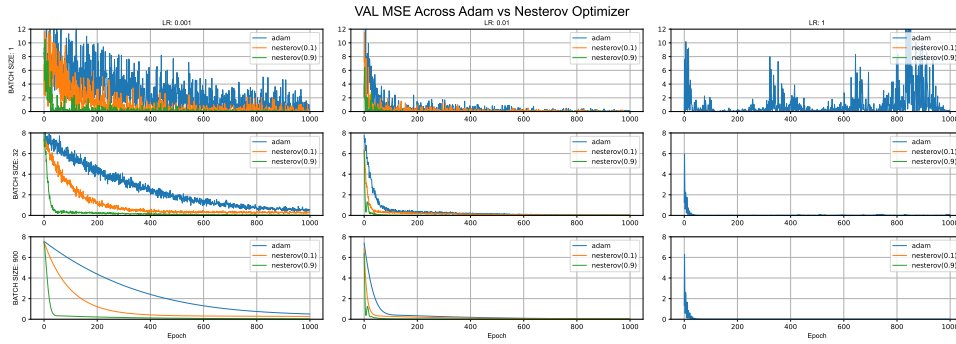


Figure 12: Validation MSE across Adam and Nesterov optimizers

From figure 12 we can observe that,

- Nesterov is performing better than Adam in almost all of the experiments.
- While Nesterov failed in learning rate 1, Adam has reached a smaller loss quickly.

4 Discussion

Based on the above experiments, we can conclude that:

- Using mini-batch gradient descent we can leverage the properties of SGD (faster update but higher noisy gradients) and Full GD (slower update but smoother gradients). i.e. tradeoff between faster updates and smoother gradients.
- Using a higher learning rate can take a bigger update step and might reach minimum loss faster but can cause gradient explosion as well. Using a lower learning rate takes a smaller update step but needs more iterations to reach minimum loss.
- Using adaptive optimizers, Adam performed better than others.
- Using fixed learning rate based optimizers, Nesterov’s performance was observed to be the best.
- When working on large-scale datasets, adaptive learning rate based optimizers are best as they update parameters with different learning rates or step lengths in each iteration for individual parameters. But can be slower due to requiring additional computations.

For our experiment, noise was selected to be small and it would be interesting to see how well loss values decrease upon increasing the noise level.

References

- [1] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.
- [2] Gao Huang et al. *Densely Connected Convolutional Networks*. 2018. arXiv: 1608.06993 [cs.CV]. URL: <https://arxiv.org/abs/1608.06993>.
- [3] Diederik P Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *Advances in Neural Information Processing Systems*. 2014.
- [4] Ning Qian. “On the momentum term in gradient descent learning algorithms”. In: *Neural Networks* 12.1 (1999), pp. 145–151. ISSN: 0893-6080. DOI: [https://doi.org/10.1016/S0893-6080\(98\)00116-6](https://doi.org/10.1016/S0893-6080(98)00116-6). URL: <https://www.sciencedirect.com/science/article/pii/S0893608098001166>.
- [5] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG].
- [6] Ilya Sutskever, James Martens, and Geoffrey E Hinton. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*. 2013.

- [7] Tijmen Tieleman. *Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. CSC321: Neural Networks and Machine Learning. 2012.
- [8] Eric W. Weisstein. *Least Squares Fitting*. MathWorld—A Wolfram Web Resource. Accessed: July 1, 2024. 2024. URL: <https://mathworld.wolfram.com/LeastSquaresFitting.html>.
- [9] Matthew D Zeiler. “Adadelata: An Adaptive Learning Rate Method”. In: *International Conference on Machine Learning*. 2012.