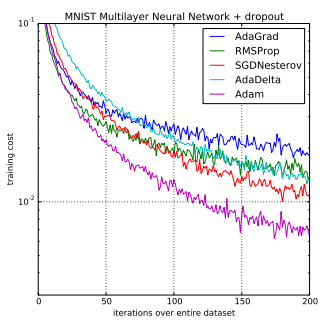
**World of optimizers.**

In the world of Machine Learning, we usually have an input dataset that can consist of millions of sample instances and our job is simple, using mathematical techniques, we must convert the given input to the desired output. After getting the algorithm in place (obtaining a model with precise parameters), we can use it to predict the output (with a level of certainty) for instances that we have not come across yet. In order to do this, we need to define a function that gives us information about the current performance of the model. In Machine Learning, we call this as the Loss function. This function has many forms, but one thing common between all these forms is that they are used to represent the current performance of the model.

However, this is not the only use of a loss function. In order for an algorithm to learn the underlying parameters to a problem, it must be trained using the input dataset and training is done in such a way that the loss of the model decreases over time (the model becomes more confident about its predictions). This can only happen if we tune the parameters used in converting the input to the output. This is where optimizers come in. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. Let us take an example of a blind hiker hiking down a hill. The hiker doesn't know where she's going, but one thing she can know for certain is whether she is going uphill (deviating from the goal) or downhill (progressing towards the goal). If she continues going downhill, she will eventually reach the base as, throughout her journey, she was progressing towards the goal.

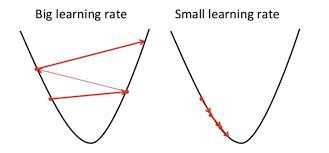
When **w** denotes the set of parameters and E(w|X) is the error with parameters w on the given training set X, we look for:

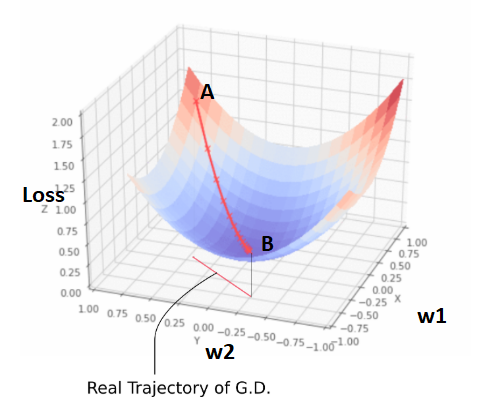
w**\*** = arg**w**min E(w|X)

Let’s explore a few of the various types of optimizers out there:

* Gradient Descent
* Adagrad
* RMSprop
* Adadelta
* Adam

1. **Gradient Descent (GD):**

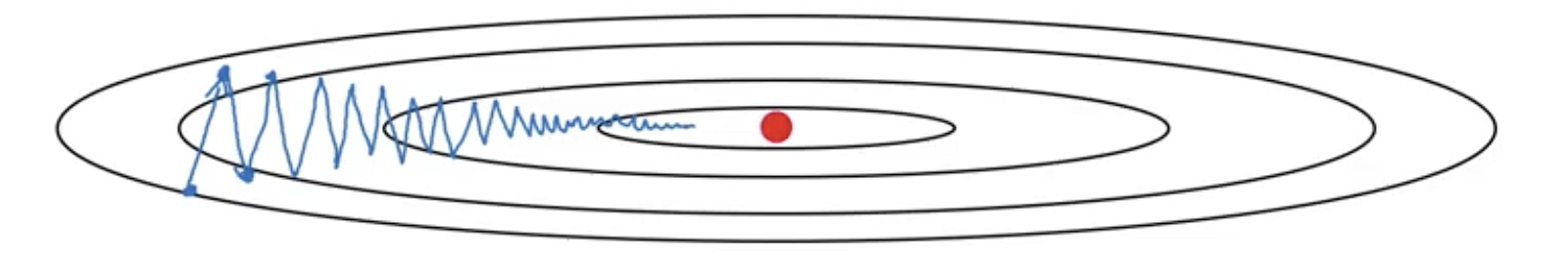
There are different types of gradient descent namely- Batch GD, Mini-batch GD, and Stochastic GD. But what is Gradient Descent? Gradient Descent is an iterative optimization technique that is used to find the values of the parameters of a model. When E(w) is the loss function and is a differentiable function of a vector of variables, we have the gradient vector composed of the partial derivatives and the gradient descent procedure to minimize E starts from a small random values initialized for w, and at each step, the algorithm updates w, in the opposite direction of the gradient (since it has to minimize the loss function). When we get to a minimum (or maximum), the derivative is 0 and the procedure terminates. This indicates that the procedure finds the nearest minimum that can be a local minimum, and there is no guarantee of finding the global minimum unless the function has only one minimum. The use of a good value for η (learning rate: a hyperparameter that is to be set by the programmer manually) is also critical; if it is too small, the convergence may be too slow, and a large value may cause oscillations and even divergence.



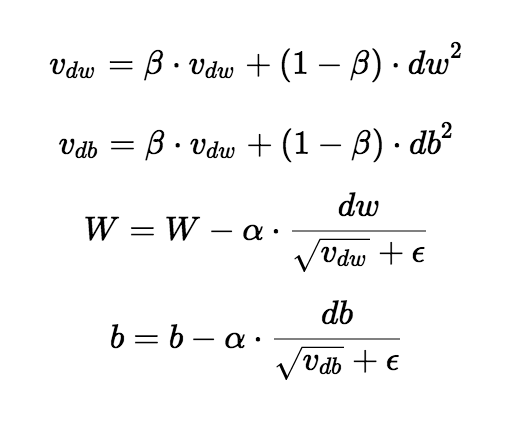
The only difference between different types of gradient descent algorithms is the manner in which the parameters are updated.

* Batch GD: The entire training set is parsed before an update is performed to the parameters.
* Mini-batch GD: The training set is divided into mini batches consisting of a subset of the input instances and no 2 mini-batches may have the same instances. The update is performed after parsing 1 mini-batch before proceeding to the next.
* Stochastic GD: The update on the parameters is performed after parsing a single instance.

2. **RMSprop: Root Mean Square Propagation**



Given above is another diagram for SGD. We can see that the standard algorithm takes larger steps in the y-direction and smaller steps in the x-direction. If we can manipulate the original equation for SGD such that we can reduce the steps taken in the y-direction and increase the steps taken in the x-direction, then the algorithm is said to converge to the goal faster. We can do this by adding a momentum term. Momentum in physics refers to the quantity and the direction of motion that an object has. Thus adding a momentum term in our standard equation enables us to “direct” the loss function towards the x-direction and simultaneously subsides the oscillations in the y-direction. The following equations show how the gradients are calculated for the RMSprop.

****

* α is the learning rate.
* β is the momentum factor, it’s a value between 0 and 1.
* W and b (weights and the biases) are the parameters for the model.
* ϵ is a small constant term added so that the term does not explode if the root value is close to 0.

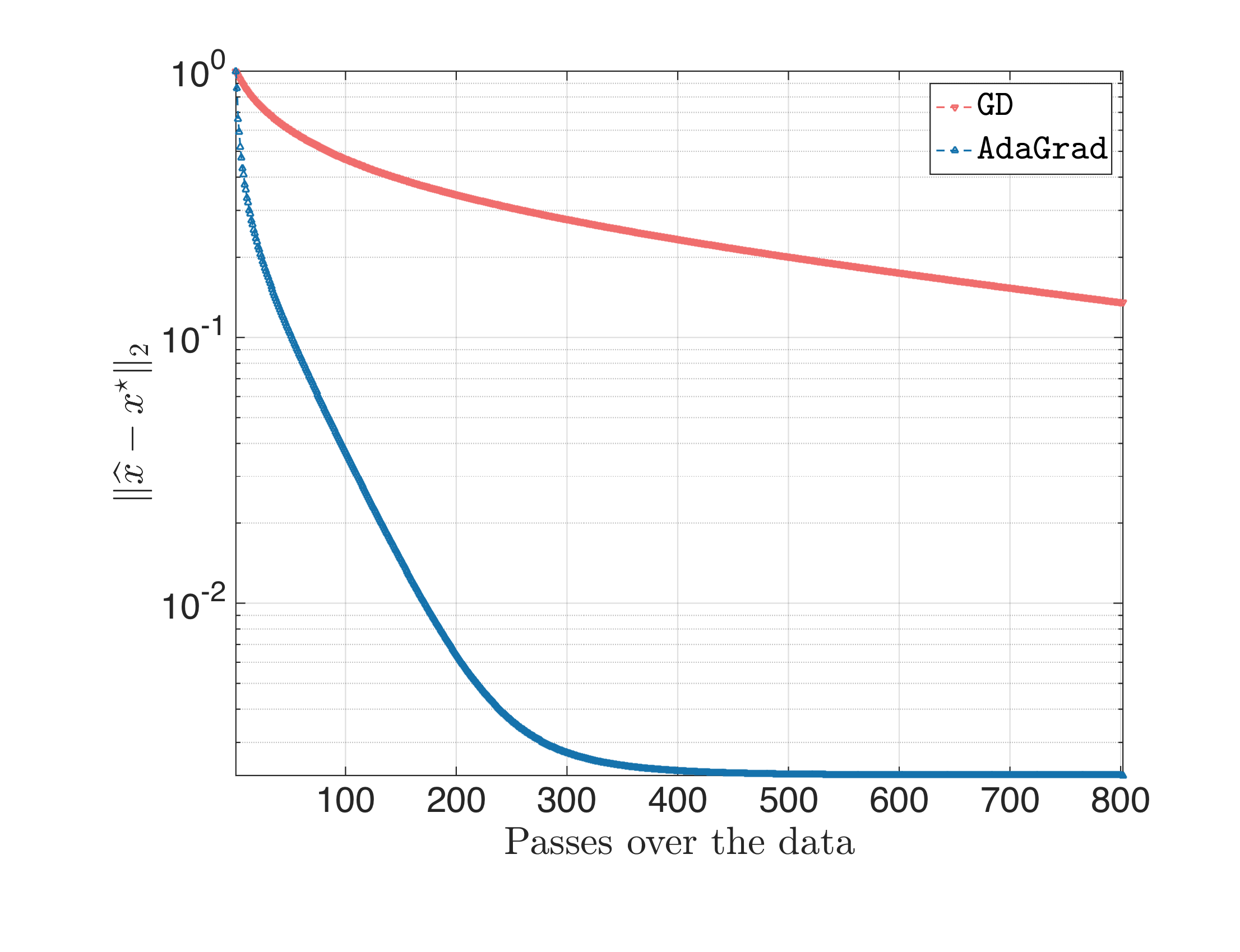
3. **Adagrad: Adaptive Gradient Algorithm**

AdaGrad is an algorithm for gradient-based optimization that adapts the learning rate to the parameters, performing smaller updates (smaller learning rates) for parameters associated with frequently occurring features, and larger updates (larger learning rates) for parameters associated with infrequent features. It is known to be well-suited while dealing with sparse data. AdaGrad greatly improved the robustness of SGD and it can be used to train industrial scale neural networks. The parameters can be updated using the following equation:

θt+1, i = θt,i- (η.gt,i)/√(Gt,ii + ϵ)

* G is a d×d (number of features of an instance = d) diagonal matrix where each diagonal entry is the sum of the squares of the gradients with respect to θi up to time step t.
* ϵ is a smoothing term that avoids division by zero (usually of the order of 1e−8).
* gt,i is then the partial derivative of the objective function with respect to the parameter θi at time step t.

One of AdaGrad's main benefits is that it eliminates the need to manually tune the learning rate. On the other hand, one disadvantage of this algorithm is that as training goes on, the square root term keeps growing. This causes the learning rate to become infinitesimally small hence it the model to cannot learn further.



4. **Adadelta:**

Adadelta also belongs to the family of stochastic gradient descent algorithms. Adagrad is sensitive to initial conditions: if the initial gradients are large, the learning rates will be low for the remainder of the training. One way to lessen this problem is to tune the global learning rate *η:* however, this cancels the advantage of Adagrad as being an adaptive learning rate algorithm.

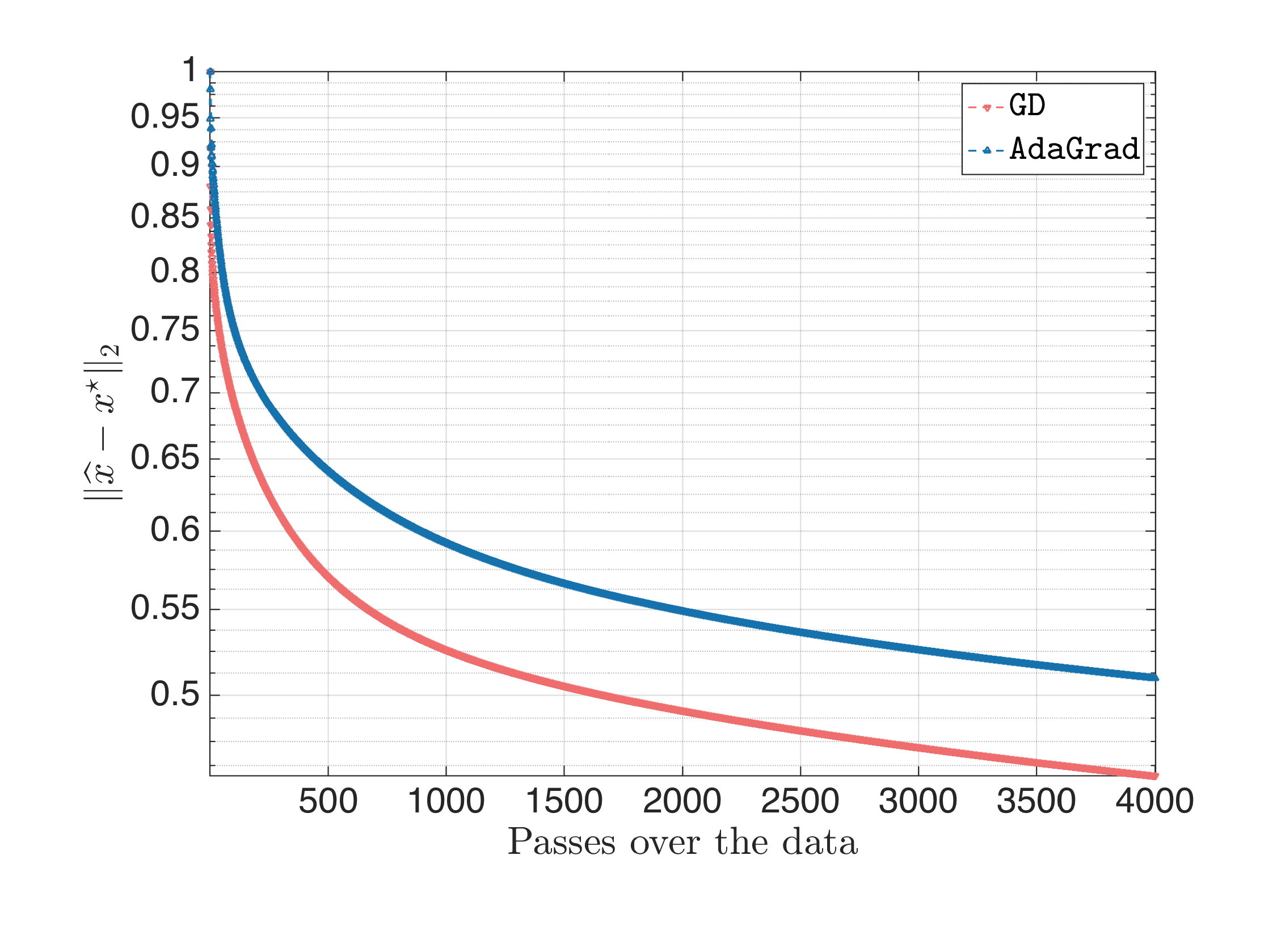
Adadelta is an extension of Adagrad that seeks to reduce its aggressive decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to a fixed size w. Instead of storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average E[g2]t at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

E[g2]t=γE[g2]t−1+(1−γ)g2t

γ, where it’s a value between 0 and 1, it decided how much of the past information affects the E[g2] at time step t.

The parameters are updated using the following equation:

θt+1= θt− (η.gt)/√(E[g2]t+ϵ)

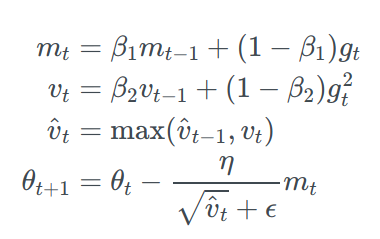


5. **Adam: Adaptive Moment Estimation**

It is used on non-convex optimization problems (problems with multiple local minima). Few of its advantages include:

* Computationally efficient.
* Little memory requirements.
* Appropriate for problems with very noisy/or sparse gradients.
* Hyper-parameters have intuitive interpretation and typically require little tuning.
* Straightforward to implement.

A learning rate is maintained for each network weight (parameter) and separately adapted as learning takes place. It contains the advantages of both, RMSprop and AdaGrad. Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance). Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters β1 and β2 control the decay rates for first moment estimates and the second moment estimates respectively.



* mt is the estimate of the first moment.
* vt is the estimate of the second moment.

Extension of Adam include:

* Adamax
* Nadam

**Adamax:**

The vt factor in the Adam update rule scales the gradient inversely proportionally to the ℓ2 norm of the past gradients (via the vt−1 term) and current gradient |gt|2..

We can generalize this update to the ℓp norm. Norms for large p values generally become numerically unstable, which is why ℓ1 and ℓ2 norms are most common in practice. However, ℓ∞ also generally exhibits stable behavior. We use ut to denote the infinity norm-constrained vt.

ut = β∞2vt−1+(1−β∞2)|gt|∞ = max(β2⋅vt−1,|gt|)

θt+1= θt− η.mt/ut

**Nadam: Nesterov-accelerated Adaptive Moment Estimation**

We incorporate Adam with the Nesterov-accelerated gradient by modifying its momentum term vt.

(Ignoring the derivation) We get the following update equation:

θt+1= θt− η.(β1.mt+(1−β1).gt /(1−β1t))/(√vt+ϵ)

**Conclusion:**

Thus we are now aware that there are several different kinds of optimizers present. We must analyze every problem carefully and determine which optimizer will be the best option. This might include making critical decisions such as whether the increased complexity and increased memory requirement are justified by the corresponding performance speedup.