CME 216, ME 343 - Spring 2020

Eric Darve, ICME

Stanford University

Adagrad is a method that attempts to adaptively change the learning rate.

It does so by using a different learning rate per parameter w_i.

Adagrad is a heuristic.

It will not yield an acceleration on all problems.

But let us see an example where Adagrad will work.

Let's consider a quadratic approximation of the loss function:

$$L(X) = \frac{1}{2} X^T H X$$

The gradient is HX.

There are some situations where the matrix H may not be properly balanced.

Remember that H is the Hessian of a general loss function L so we have no control over its properties.

One way to think of a matrix that is not balanced is to define some $0 < \beta < 1$ and

$$[D_\beta]_{ii} = \beta^{i-1}, \quad i = 1, \dots, n$$

Then assume that our Hessian is H_β with:

$$H_\beta = D_\beta H D_\beta$$

H is a symmetric positive definite matrix with a condition number close to 1.

$$H_\beta = D_\beta H D_\beta$$

As β goes to 0, the matrix becomes increasingly imbalanced.

Some of the rows/columns become very small.

Gradient methods will typically converge fast with H.

However, they will struggle with H_β because it is ill-conditioned due to the scaling matrix D_β.

Adagrad is able to overcome some of the difficulties with H_β.

$$\Delta X = -\alpha H_\beta X$$

Δx_n becomes very small as $\beta \to 0$.

We have seen previously how we could use an eigendecomposition of H_β and look at the convergence of individual modes

$$Z = U^T X$$

$$\Delta Z = -\alpha \Lambda Z$$

Remember how we said that ideally $\alpha = \lambda_i^{-1}$.

In this case, we can do something close.

We will use the adaptive learning rates of Adagrad.

$$H_\beta = U \Lambda U^T$$

We are not going to prove this, but because of the scaling matrix D we have that:

- U is close to identity
- $\lambda_i \propto \beta^{2(i-1)}$

As predicted, some of the modes are going to converge very slowly with a conventional learning rate.

Pick $\alpha = \lambda_1^{-1}$. Mode n is updated using

$$\Delta z_n = -\frac{\lambda_n}{\lambda_1} z_n = -\beta^{2(n-1)} z_n$$

This is very slow.

Let's consider again the update equation for X:

$$\Delta X = -\alpha U \Lambda U^T X$$

What can we do without computing H and its eigendecomposition (U, Λ)?

There is an approximate but simple strategy:

$$\left|\frac{\partial L}{\partial x_i}\right| = |[HX]_i| \approx \beta^{i-1} \, \|X\|_2$$

We could choose as a learning rate for component i:

$$\alpha \leftarrow \frac{\alpha}{\left|\frac{\partial L}{\partial x_i}\right|}$$

That would not really work because $\nabla_W L \to 0$.

Adagrad uses the following formula instead:

$$\boldsymbol{s}_i = \sum_k \left(\frac{\partial L_k}{\partial x_i} \right)^2$$

where k is the batch index.

The update rule is then

$$\Delta x_i = -\frac{\alpha}{\sqrt{s_i + \epsilon}} \frac{\partial L_k}{\partial x_i}$$

The ϵ is a regularizing factor that accounts for cases where s_i may become too small, which can happen at the beginning, in some rare cases.

Although the strategy is simple, it provides a substantial acceleration in our case.

The formula is

$$\Delta x_i = -\frac{\alpha}{\sqrt{s_i + \epsilon}} \frac{\partial L}{\partial x_i} = -\frac{\alpha}{\sqrt{s_i + \epsilon}} [HX]_i$$

Using matrix notation,

$$X^{(k+1)} - X^{(k)} = -\alpha DHX^{(k)}$$

$$X^{(k+1)} = (I - \alpha DH)X^{(k)}$$

We can solve for step k in terms of the initial value:

$$X^{(k)} = (I - \alpha DH)^k X^{(0)}$$

where D is a diagonal matrix with

$$d_{ii} = \frac{1}{\sqrt{s_i + \epsilon}}$$

In Adagrad, $\boldsymbol{s}_i$ can increase quite a bit.

Recall that:

$$\boldsymbol{s}_i = \sum_k \left(\frac{\partial L_k}{\partial x_i} \right)^2$$

If convergence is slow, $\boldsymbol{s}_i$ grows.

We will fix this problem with RMSProp.

If convergence is reasonably fast (or we use RMSProp), we can approximate $\boldsymbol{s}_i$ by

$$\boldsymbol{s}_i \propto \beta^{2i}$$

In that case, we can prove the following result.

The largest eigenvalue of $I - \alpha H$ is

$$\approx 1 - \beta^{2(n-1)}$$

while for Adagrad with $I - \alpha DH$, it is

$$\approx 1 - \beta^{n-1}$$

This implies that the number of iterations has been reduced by

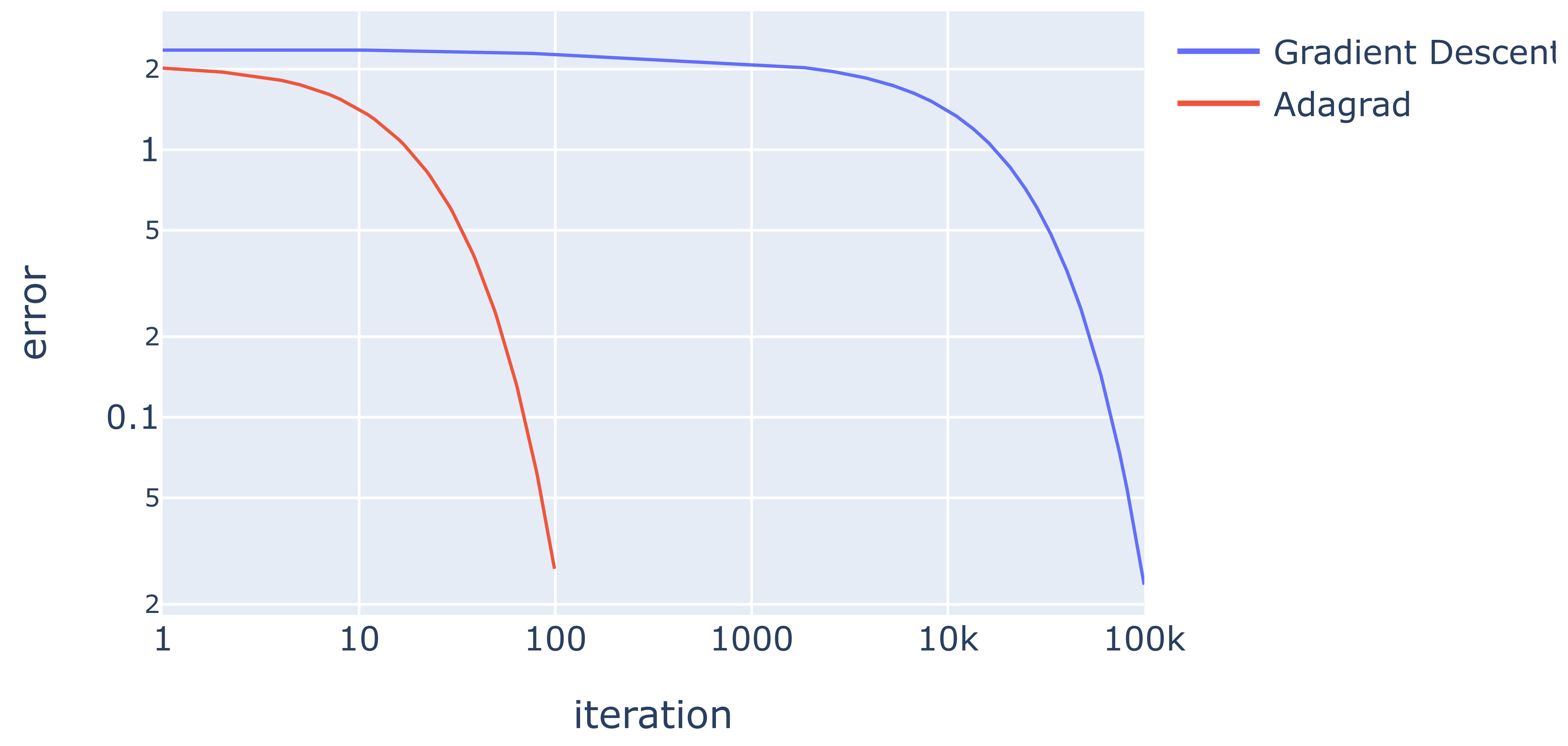
$$\left(\frac{1}{\beta}\right)^{n-1}$$

If $\beta = 0.9$ and $n = 100$, that number is close to 30k.

We ran a small benchmark in the notebook.

We picked: $\beta = 0.1$ and $n = 4$.

The speedup is theoretically about 1,000 in this case.



error
2
1
5
2
0.1
5
2
1
10
100
1000
10k
100k
iteration
Gradient Descent
Adagrad

Of course in practice things are not as simple.

We can get slow convergence even if H is perfectly balanced and all components of the gradients are of similar magnitudes.

But there are many applications where some components of the gradient are systematically smaller than over components.

Adagrad will improve convergence in these cases.

```python
def adagrad(W, s, lr, batch_size):
    eps_stable = 1e-7
    g = W.grad / batch_size
    s += square(g) # element-wise square
    W -= lr * g / sqrt(s + eps_stable) # element-wise division
```