CME 216, ME 343 - Spring 2020

Eric Darve, ICME

Adagrad can yield nice acceleration in some cases but its formula to scale the learning rate has some issues:

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

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

If the convergence is slow, the gradient decays slowly.

As a result, s_i grows, which results in small steps and further slow down in convergence.

To address this issue, RMSProp uses the idea of momentum so that the scaling factor has a more controlled growth:

$$\boldsymbol{s}_i \leftarrow \beta \boldsymbol{s}_i + (1-\beta)\left(\frac{\partial L_k}{\partial \boldsymbol{x}_i}\right)^2$$

$$\Delta \boldsymbol{x}_i = -\frac{\alpha}{\sqrt{\boldsymbol{s}_i} + \epsilon} \frac{\partial L_k}{\partial \boldsymbol{x}_i}$$

When β is small, only recent values of the gradient will contribute.

When β is close to 1, s_i is close to

$$\approx \frac{1}{k_0} \sum_{s=0}^{k_0} \left(\frac{\partial L_{k-s}}{\partial x_i} \right)^2$$

Another important element of RMSProp is that it helps around saddle points.

Near saddle points, the gradient $\nabla_x L$ becomes very small.

By scaling with $1/\left(\sqrt{\boldsymbol{s}_i} + \boldsymbol{\epsilon}\right)$, we can get a nice boost.

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

The last method we will cover is Adam. It addresses two limitations of RMSProp.

The first one is that it adds momentum to the gradient calculation as well.

So momentum is applied to two places: to advance the gradient squared and the gradient itself.

$$m_i \leftarrow \beta_1 m_i + (1 - \beta_1) \frac{\partial L_k}{\partial x_i}$$

$$s_i \leftarrow \beta_2 s_i + (1 - \beta_2) \left(\frac{\partial L_k}{\partial x_i} \right)^2$$

The second modification has to do with initialization.

The problem with momentum is that before batch 0, all variables are initialized to 0.

As a result, we tend to get small values initially.

Basically initially we only see

$$m_i \approx (1 - \beta_1) \frac{\partial L_k}{\partial x_i}$$

instead of

$$m_i \approx \frac{\partial L_k}{\partial x_i}$$

This can be made more precise with the following derivation.

We will find how to rescale m_i to avoid this problem in the simple case where we assume that the gradient is constant.

It's not perfect but it will improve the initialization bias quite a bit.

Let's assume that the gradient is constant.

In that case we just want

$$m_i = \frac{\partial L_k}{\partial x_i}$$

at all steps.

Let's compare with what we are actually getting.

Denote G_i the gradient and assume it is constant.

$$m_i \leftarrow \beta_1 m_i + (1 - \beta_1) G_i$$

If we solve this equation we get, at step k:

$$m_i^{(k)} = \beta_1^k m_i^{(0)} + (1 - \beta_1) G_i \sum_{l=0}^{k-1} \beta_1^l$$

We chose $m_i^{(0)} = 0$. So, we get:

$$\sum_{l=0}^{k-1} \beta_1^l = \frac{1 - \beta_1^k}{1 - \beta_1}$$

$$m_i^{(k)} = (1 - \beta_1^k) G_i$$

The scaling factor we pick is therefore:

$$\frac{1}{1 - \beta_1^k}$$

when processing batch k.

The final formulas are.

Gradient and gradient-squared:

$$m_i \leftarrow \beta_1 m_i + (1 - \beta_1) \frac{\partial L_k}{\partial x_i}$$

$$s_i \leftarrow \beta_2 s_i + (1 - \beta_2) \left(\frac{\partial L_k}{\partial x_i} \right)^2$$

Initialization bias:

$$\hat{\boldsymbol{m}}_i \leftarrow \frac{\boldsymbol{m}_i}{1-\beta_1^k}$$

$$\hat{\boldsymbol{s}}_i \leftarrow \frac{\boldsymbol{s}_i}{1-\beta_2^k}$$

Gradient step:

$$\Delta x_i = -\frac{\alpha}{\sqrt{\hat{s}_i} + \epsilon} \hat{m}_i$$

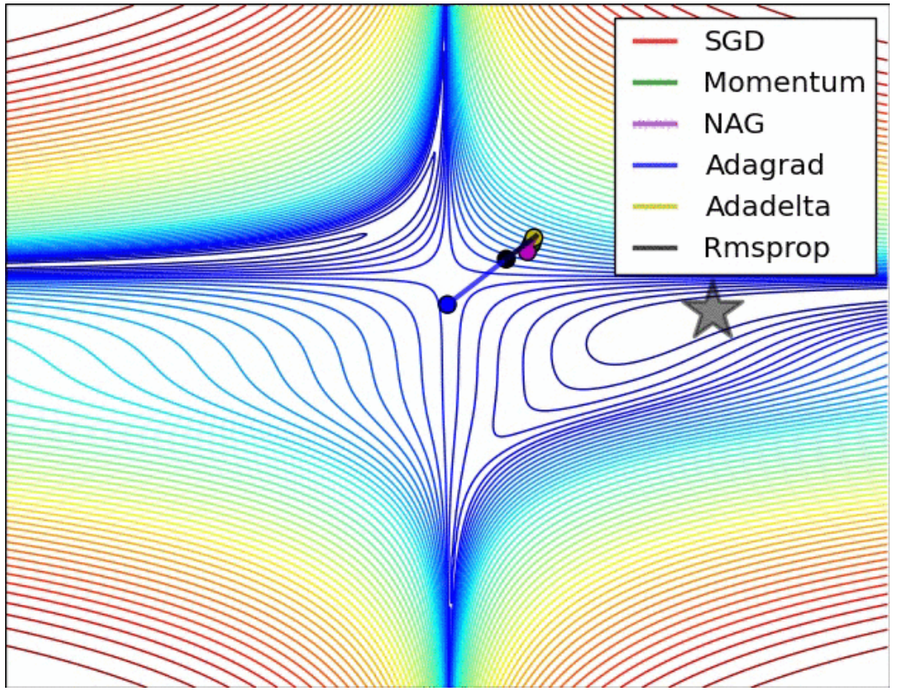
β_1 is for the gradient momentum. It is typically chosen equal to 0.9.

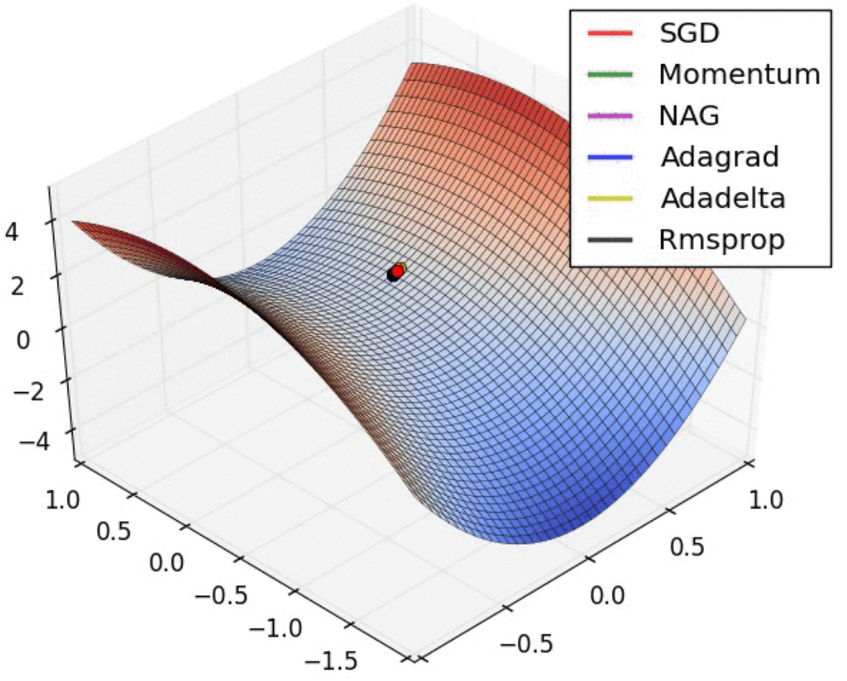
β_2 is for the gradient squared momentum. This one is chosen very close to 1, 0.999. This means that $\boldsymbol{s_i}$ changes only very slowly.

The number of steps required for changes in $\boldsymbol{s_i}$ to be visible is on the order of 1,000 steps.

```
def adam(W, m, s, lr, batch_size, k):
    beta1 = 0.9
    beta2 = 0.999
    eps_stable = 1e-8

    g = W.grad / batch_size

    m = beta1 * m + (1. - beta1) * g
    s = beta2 * s + (1. - beta2) * square(g)

    m_hat = m / (1. - beta1 ** k)
    s_hat = s / (1. - beta2 ** k)

    W -= lr * m_hat / (sqrt(s_hat) + eps_stable)
```

You can check these animations on the CS231n class page.





Momentum has a β close to 1, and tends to overshoot too much.

Adagrad and RMSProp scale the gradient near the saddle point which provides a nice boost.

Additional reading

- Adaptive Subgradient Methods for Online Learning and Stochastic Optimization by Duchi et al., 2011
- Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, International Symposium on Mathematical Programming 2012, by Duchi et al.

- Qualitatively characterizing neural network optimization problems by Goodfellow et al., ICLR, 2015
- Practical Recommendations for Gradient-Based Training of Deep Architectures by Bengio, 2013
- Efficient Backprop by LeCun et al., 2012
- ADADELTA: an adaptive learning rate method by Zeiler, 2012