DS 4440  Optimizer Matters

Gradient Descent

\[ \hat{\theta}_{t+1} = \hat{\theta}_t - \alpha \nabla_{\theta} L(\hat{\theta}_t, x, y) \]

either per instance or batch

Variants

- Stochastic: One instance at a time
- Batch: All instances at once
- Mini batch: b samples at once

Basic GD is myopic which can cause Zig-Zagging \( \rightarrow \) Slow convergence

Sometimes loss curves are sharper in some dims than others.

Normalizing inputs can mitigate this a bit.
Momentum is another approach

\[ \nabla_t \leftarrow \gamma \nabla_{t-1} - \alpha \nabla \ell (\hat{\theta}_{t-1}, x, y) \]

Keep going in direction you've been going, but adjust locally:

\[ \hat{\theta}_t \leftarrow \hat{\theta}_{t-1} + \nabla_t \]

In fact, we already "know" where we are going at this point, why not look ahead? This is the idea behind Nesterov gradient descent, a variant of momentum:

\[ \nabla_t \leftarrow \gamma \nabla_{t-1} - \alpha \nabla \ell (\hat{\theta}_{t-1} + \gamma \nabla_{t-1}, x, y) \]

Move toward accumulated gradient.

(See examples in Colab.)

So far we have assumed the learning rate is the same for all parameters. But you can imagine optimal rates varying across parameters.
Let: \( g_{t,j} = \nabla_{\theta} L(\Theta_t)_j \)

That is, the partial for param \( j \) at iteration \( t \). So far we have said:

\[
\hat{\Theta}_{(t+1),j} \leftarrow \hat{\Theta}_{t,j} + \alpha g_{t,j}
\]

But having a global \( \alpha \) may be suboptimal.

**Adagrad** is one approach that defines adaptive, per-parameter learning rates.

\[
\hat{\Theta}_{(t+1),j} \leftarrow \hat{\Theta}_{t,j} + \alpha_{t,j} g_{t,j}
\]

**Intuition:** Move slower for parameters we have updated a bunch already.

\[
\alpha_{t,j} \overset{\text{def}}{=} \frac{\alpha}{G_{t,j} + \epsilon}
\]

Where \( G_{t,j} \) is the sum of the squares of grads for param \( j \).

This avoids div. by \( \phi \) by

\[
\sum_{t} \nabla_{\theta} (L(\hat{\Theta}_{t,x}, y))_j^2
\]

\[
\hat{\Theta}_{(t+1),j} \leftarrow \hat{\Theta}_{t,j} - \alpha_{t,j} g_{t,j}
\]
If we have accumulated a lot of updates for $j$ by time $t \rightarrow$ Smaller steps

All entries in $G$ grow over time; this means descent slows.

**RMSProp** Extends Adagrad by reducing the effect of monotonically $\downarrow$ learning rates. We keep a decaying average of past Squared gradients.

\[
E[g^2]_t \overset{\text{def}}{=} \lambda E[g^2]_{t-1} + (1-\lambda)g^2_t
\]

Where $\lambda$ is our decay term.

Can replace entries in $G$ with this. This is very similar to AdaDelta.

**Adam** (Adaptive Moment Estimation)

Also maintains parameter-wise learning rates
\[ m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \]
\[ v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g^2_t \]

Then:
\[ \hat{\Theta}_{(t+1), j} \leftarrow \hat{\Theta}_{t, j} - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot m_t \]

This is a bit like Momentum + RMSProp

**Optimizers Recap**

Lots of options; a few key ideas

- Exploiting history
- Per-parameter and adaptive rates
A few notes on regularization & overfitting

Weight regularization

Impose a norm loss to keep weights small.

Dropout

Randomly mask (drop) weights during training.

\[ V = \text{random sample } [0, 1] \rightarrow V_j = 0 \text{ if } V_j < p \text{ else } 1 \]

\[ \hat{h} = h \odot V \]

At test time: \[ \hat{h} = h \odot V \]

Early Stopping

Monitor loss on a ‘nested’ set; keep \( \hat{O} \) that gave best performance.