Automatic Differentiation (AD)

- A method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards
  - Takes code that computes a function and uses that to compute the derivative of that function
  - The goal isn’t to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Differentiation and Programming

Example (Math)

\[
\begin{align*}
x &= ? \\
y &= ? \\
a &= xy \\
b &= \sin(x) \\
z &= a + b
\end{align*}
\]

Example (Code)

\[
\begin{align*}
x &= ? \\
y &= ? \\
a &= x \times y \\
b &= \sin(x) \\
z &= a + b
\end{align*}
\]
Forward Mode AD

- To translate using the rules we simply replace each primitive operation in the original program by its differential analogue

- The order of computation remains unchanged: if a statement K is evaluated before another statement L, then the differential analogue of K is evaluated before the analogue statement of L

- This is *Forward-mode Automatic Differentiation*

Backward AD:
Reversing the Chain Rule

\[
\frac{\partial s}{\partial u} = \sum_i \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i}
\]

\[
x = ? \\
y = ? \\
a = x \cdot y \\
b = \sin(x) \\
z = a + b
\]

\[
\frac{\partial s}{\partial z} = ?
\]

\[
\begin{align*}
\frac{\partial s}{\partial b} &= \frac{\partial b}{\partial z} \frac{\partial s}{\partial b} \\
\frac{\partial s}{\partial a} &= \frac{\partial a}{\partial z} \frac{\partial s}{\partial a} \\
\frac{\partial s}{\partial y} &= \frac{\partial y}{\partial a} \frac{\partial s}{\partial y} \\
\frac{\partial s}{\partial x} &= \frac{\partial x}{\partial a} \frac{\partial s}{\partial x} + \frac{\partial b}{\partial x} \frac{\partial s}{\partial b} \\
&= y \frac{\partial s}{\partial a} + \cos(x) \frac{\partial s}{\partial b} \\
&= (y + \cos(x)) \frac{\partial s}{\partial z}
\end{align*}
\]
Visualizing Dependencies

• Differentiating in reverse can be quite mind-bending: instead of asking what input variables an output depends on, we have to ask what output variables a given input variable can affect.

• We can see this visually by drawing a dependency graph of the expression:

\[
\begin{align*}
\sin \quad x \quad \cdot \\
\quad b \quad a \\
+ \quad z \\
\end{align*}
\]

Gradient Descent

• Define total loss as \( \mathcal{L} = \sum_{(x,y) \in D} \ell(g(x, \theta), y) \) for some loss function \( \ell \), dataset \( D \), and model \( g \), with learnable parameters \( \theta \)

• Define how many passes over the data to make (each one known as an Epoch)

• Define a learning rate \( \eta \)

Gradient Descent updates the parameters \( \theta \) by moving them in the direction of the negative gradient with respect to the total loss \( \mathcal{L} \) by the learning rate \( \eta \) multiplied by the gradient:

for each Epoch:

\[
\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}
\]
Gradient Descent

- Gradient Descent has good statistical properties (very low variance)
- But is very data inefficient (particularly when data has many similarities)
- Doesn’t scale to effectively infinite data (e.g. with data augmentation)

Stochastic Gradient Descent (SGD)

- Define loss function $\ell$, dataset $D$, and model $g$, with learnable parameters $\theta$
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate $\eta$

Stochastic Gradient Descent (SGD) updates the parameters $\theta$ by moving them in the direction of the negative gradient with respect to the loss of a single item $\ell$ by the learning rate $\eta$, multiplied by the gradient:

$$\text{for each Epoch: for each } (x, y) \in D: \quad \theta \leftarrow \theta - \eta \nabla_{\theta} \ell$$
Stochastic Gradient Descent (SGD)

- Stochastic Gradient Descent has poor statistical properties (very high variance)
- Why works?
  - We don't need to check all the training examples to get an idea about the direction of decreasing slope. By analyzing only 1 example at a time and following its slope (gradient), we can reach a point very close to the actual minimum
- Not computationally efficient enough (poor utilization of resources w.r.t. vectorization)

Mini-Batch Stochastic Gradient Descent

- Define a batch size $b$
- Define batch loss $\mathcal{L}_b = \sum_{(x,y) \in D_b} \ell(g(x, \theta), y)$ as for some loss function $\ell$ & model $g$ with learnable parameters $\theta$. $D_b$ is a subset of dataset $D$ of cardinality $b$
- Define how many passes (Epochs) over the data to make
- Define a learning rate $\eta$

Mini-batch Stochastic Gradient Descent (SGD) updates parameters $\theta$ by moving them in the direction of the negative gradient with respect to the loss of a mini-batch $D_b$, $\mathcal{L}_b$ by the learning rate $\eta$, multiplied by the gradient: partition the dataset $D$ into an array of subsets of size $b$

for each Epoch:

for each $D_b \in partitioned(D)$:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} \mathcal{L}_b$$
Mini-Batch Stochastic Gradient Descent

- Mini-batch Stochastic Gradient Descent has reasonable statistical properties (much lower variance than SGD)
- Allows for computationally efficiency (good utilization of resources)
- Ultimately we would normally want to make our batches as big as possible for lower variance gradient estimates, but:
  - Must still fit in RAM (e.g. on the GPU)
  - Must be able to maintain throughput (e.g. pre-processing on the CPU; data transfer time)

Learning Rates

- Choice of learning rate is extremely important
- But we have to reason about the ‘loss landscape’
  - Most convergence analysis of optimization algorithms assumes a convex loss landscape
    - Easy to reason about
    - Can be shown that (S)GD will converge to the optimal solution for a variety of learning rates
    - Can give insights into potential problems in the non-convex case
  - Deep Learning is highly non-convex
    - Many local minima
    - Plateaus
    - Saddle points
    - Symmetries (permutation, etc)
    - Certainly no single global minima
Accelerated Gradient Methods

- Accelerated gradient methods use a *leaky* average of the gradient, rather than the instantaneous gradient estimate at each time step.

- A physical analogy would be one of the momentum a ball picks up rolling down a hill...

- This helps address the *GD failure modes, but also helps avoid getting stuck in local minima.

Momentum I

- It’s common for the ‘leaky’ average (the ‘velocity’, $v_t$) to be a weighted average of the instantaneous gradient $g_t$ and the past velocity$^1$:

$$v_t = \beta v_{t-1} + g_t$$

where $\beta \in [0, 1]$ is the ‘momentum’

$^1$There are quite a few variants – here the PyTorch variant.
Momentum II

- The momentum method allows to accumulate velocity in directions of low curvature that persist across multiple iterations.

- This leads to accelerated progress in low curvature directions compared to gradient descent.

MB-SGD with Momentum

- Learning with momentum on iteration $t$ (batch at $t$ denoted by $b(t)$) is given by:

$$
\begin{align*}
\mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \nabla \theta \mathcal{L}_{b(t)} \\
\theta_t &\leftarrow \theta_{t-1} - \eta \mathbf{v}_t
\end{align*}
$$

$\beta = 0.9$ is a good choice for the momentum parameter.
SGD with Momentum - potentially better convex convergence

Learning Rates

- In practice you want to decay your learning rate over time
- Smaller steps will help get closer to the minima
- But don't do it too early, else might get stuck
- Something of an art form!
**Reduce LR on Plateau**

- Common Heuristic approach:
  - if the loss hasn’t improved (within some tolerance) for \( k \) epochs then drop the LR by a factor of 10
- Remarkably powerful!

**Cyclic Learning Rates**

- Worried about getting stuck in a non-optimal local minima?
- Cycle the learning rate up and down (possibly annealed), with a different LR on each batch
- See https://arxiv.org/abs/1506.01186
More Advanced Optimizers

- **Adagrad**
  - Decrease learning rate dynamically per weight.
  - Squared magnitude of the gradient (2nd moment) used to adjust how quickly progress is made - weights with large gradients are compensated with a smaller learning rate.
  - Particularly effective for sparse features.

- **RMSProp**
  -Modify Adagrad to decouple learning rate from gradient magnitude scaling
  - Incorporates leaky averaging of squared gradient magnitudes
  - LR would typically follow a predefined schedule

- **Adam**
  - Essentially takes all the best ideas from RMSProp and SDG+Momentum
  - Bias corrected momentum and second moment estimation
  - It might still diverge (or be non optimal, even in convex settings)...
  - LR is still a hyperparameter (you might still schedule)

Take-away Messages

- The loss landscape of a deep network is complex to understand (and is far from convex)
- If you’re in a hurry to get results use Adam
- If you have time, then use SGD (with momentum) and work on tuning learning rates
- If you’re implementing something from a paper, then follow what they did!

For more about Numerical Optimization: CMSC 764