Lecture 5

Stochastic Gradient Descent

- Vanilla GD is not popular b/c
  - full gradient is expensive to compute
  \[ \mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} L(f(x_i; \theta), y_i) \]
  
  \[ \frac{\partial \mathcal{L}}{\partial \theta} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L_i}{\partial \theta} \]
  
  - have to sum over full dataset for every weight update
  
- can easily get stuck in a local minimum
SGD solves both problems in some way.

**Idea:** allow gradients to be noisy

How?

Compute gradients over subset of data.

- Pure SGD: one datapoint: $\frac{\partial L}{\partial w} = -y \frac{\partial L}{\partial w}$
- Minibatch SGD: chunk of data: $\frac{\partial L}{\partial w} = -y \frac{\partial L}{\partial w}$

Preferred in practice: computational efficiency (parallel over minibatches)

"1 epoch" after each epoch, shuffle data!
Example: MNIST Deep Learning

- Data: 60,000 images
- Weights: 600,000
- Mini-batch size: 128
- 60k steps per epoch

Variations on (S)GD:

- Momentum: can get stuck
- Shale: idea of momentum: give gradients a push based on their previous history

Note: GD can get in trouble here
**GO 1/ moment:**  
\[ \mathbf{v}_t = \beta \mathbf{v}_{t-1} + \gamma \nabla \mathbf{L} \]

**W → W - v_t**

---

**Adaptive learning rates:**

- **Adagrad:**  
  \[ \mathbf{w} \rightarrow \mathbf{w} - \frac{\nabla \mathbf{L}}{\sqrt{G}} \]

- **G** in time

  - large gradients → smaller learning rate
  - accumulating gradients → decrease learning rate

- **Adadelta:** Adagrad w/ finite time window

  \[ G = \sum (\nabla \mathbf{L})^2 \text{ over time windows} \]

- **Adam (Adaptive Moment estimate):** Adadelta w/ momentum

  \[ \hat{v}_t = \beta \hat{v}_{t-1} + (1 - \beta) \nabla \mathbf{L} \]

  \[ \mathbf{w} \rightarrow \mathbf{w} - \frac{\hat{v}_t}{\sqrt{\hat{v}_t} + \epsilon} \]

https://rutgersconnect-my.sharepoint.com/personal/shih_physics_rutgers_edu/_layouts/15/Doc.aspx?sourcedoc={1c8418de-3e15-43ad-8ded-ca103e6850f}&action=edit&wd=target%28Physics 694 Lecture 5.one%7C9653... 2/2
Backpropagation: how gradients are computed efficiently.  

FF NN's have a recursive structure.  

\[ f(x) = A(w_h \cdot A(w_{h-1} \cdot \ldots A(w_1x)) \ldots ) \rightarrow \]

\[ x = A(w_h x_h) \]

\[ x_{h-1} = A(w_{h-1} x_{h-1}) \]

\[ \vdots \]

\[ x_2 = A(w_2 x_2) \]

\[ x_1 \equiv x. \]

\[ L = \sum L(f(x_i), y) \]

\[ \frac{\partial L}{\partial w_k} \rightarrow \frac{\partial L}{\partial w_k} \rightarrow \frac{\partial f(x_i;w)}{\partial w_k} \]

Let's work out a few examples:

\[ \frac{\partial f}{\partial w_h} = A'(w_h x_h) \cdot x_h \equiv G_h x_h \]

\[ \frac{\partial f}{\partial w_{h-1}} = A'(w_h x_h) \cdot w_h A'(w_{h-1} x_{h-1}) x_{h-1} \equiv G_{h-1} x_{h-1} \]

\[ \frac{\partial f}{\partial w_{h-2}} = A'(w_h x_h) \cdot w_h A'(w_{h-1} x_{h-1}) w_{h-1} A'(w_{h-2} x_{h-2}) x_{h-2} \equiv G_{h-2} x_{h-2} \]

\[ \vdots \]

- Compute gradients recursively.
Backpropagation: \[ G_{k-1} = G_k u_k A (w_{k-1} x_{k-1}) \]

- Matrix mult. - expensive
- enable "end-to-end"
  - training of NNS.

GPUs are key for deep learning.