Bryant Har

Lec 1-5

- Numpy is fast (not asymptotically better)
- Backprop is faster than numerical. Uses chain rule. Consumes lot of memory to store DAG computation trees.
  - Forward AD. Chain rule from dw/dw to df/dw
  - Reverse AD. Chain rule from df/df to df/dw
- GPUs/Hardware is better.
  - eager (compute values & manifest graph at the same time). pytorch
  - lazy (manifest graph first, then compute values). tensorflow

Lec 6-7

- Newton's method is $O(nd^2 + d^3)$, $O(d^2)$ memory
- GD assuming lipschitz L and strong convexity μ converges if $\alpha L \leq 2$. Average gradient for all examples and $w_t = w_{t-1} - \nabla f(w)$ for loss/ERM $f(w)$.
  - Runtime: $O(n\kappa \ln(1/\epsilon))$
  - $\kappa = L/\mu$ is condition number. Lower is easier problem to solve. Measure of difference between upper and lower bounds of steepness.
- Convergence rate guaranteed $O(1/\kappa)$ if $\alpha L \leq 1$.
  - If not conditioned as above, steep gradients can shoot weights randomly and prevent convergence. Still usually works well in practice.
- SGD. Does same thing but only use random batch B instead of whole gradient.
  - Converges to noise ball in expectation. Much faster.
  - $O(\kappa B \ln(1/\epsilon))$
  - Momemtum : $O(n\sqrt{\kappa} \ln(1/\epsilon))$

Lec 8-10

- Momentum good. Usually can reduce $\kappa \to \sqrt{\kappa}$. High κ leaves uneven level sets of the loss.
- Preconditioning reduces κ. Premultiply gradient by diagonal matrix P. Allows you to even out the level sets. Changes the effective optimization problem/plane.
- Polyak Momentum - heavy ball. Add some of previous iter momentum.
- Nesterov Momentum - Exponential moving average of previous momentum.
  Step Size Schemes
- $1/t$ step size scheme is common. $\alpha_t \propto 1/t$ for t iterations.
- Adagrad - use running simple moving average of past gradients divided by their inverse square root $\sum 1/\sqrt{d_i}$. Decreases effective step size in flat regions (low grad) and vice versa.
- RMSProp - Same thing but use exponential moving average
- Adam - Same as RMSProp but use momentum and debias sum with β.

Lec 11

- SVRG is op. Significantly decreases convergence time/iterations to SGD.
  - Calculate full GD step periodically and then run a series of SGD steps
  - Faster than SGD since fewer iterations.
- Polyak Averaging - Taking average of the noise ball and gradients gets closer to the true optimum.
  - Requires a warm-up period to get a good initial sample of the gradients.

Lec 12

- Previous reduces n cost. Reduce dimension d using
  - Random projection
    - Only care about creating a partial order between points. Just randomly project onto a line.
  - Principal Component Analysis
    - Use eigenvectors to get a better line to project onto. Best line is a linear combo of $\lambda_1 v_1 + \ldots$.
  - Autoencoders
    - Train encoder NN block to map input to smaller dimension. Train decoder NN block to map input back to initial dimension. Make the input and final output as similar as possible
  - Sparse matrices and CSRs/CSCs are useful data formats if matrices are sparse (low density of nonzero entries). Easy to compute differences between examples. Keep track of nonzero entry coords and values.

Lec 13-15

- Neural Networks involve
  - Residual NN. feedback connections. feed into self.
  - Convolutional NN. Linear layers are all a subset of a set of convolutions and filter layers.
  - Recurrent NN: Repeat running through some layers to process sequential/structured data
    - Problems: Vanishing grad, exploding grad, long-range dependencies, hard to parallelize
  - Transformers - Use attention blocks to do RNN (above) in parallel.

Transformers pipeline

Map string to tokens.

Map Tokens to embeddings/encodings. (positional encoders here or in MHA)

Layer Norm Layer

Multi-Head Attention Block (Important)

- Use masks to zero out future token attentions in softmax with -inf (preserve causality)
- Q, K, V - Query, Key, Value input activation matrices. This is **LEARNED**
  - Compute attentions as $MaskedAttention = softmax(\frac{QK}{\sqrt{dimension}} + mask)V$
  - MHA is h attention layers in parallel along head dim.

- $MHA(XW_Q^T, XW_K^T, XW_V^T)W_O^T$ (W_O is an irrelevant regularization term)
  Overall, processes interactions between tokens

Residual feedback to input

Layer Normalization

Multi-Layer Perceptron (MLP)

Done parallel wrt to each token. Each token (or group of tokens) is processed independently.

Residual feedback back to MHA output

Final Output Probability Distribution

Probability distribution is on $\mathbb{R}^{vocabulary}$

Stack more transformer blocks (optional)

Lec 15

- Batch Normalization Layer.
  - Normalize batch inputs before feeding into a layer. Usually faster convergence and mitigates exploding gradients and heavier early weighting.
- Each element = $\gamma\frac{u-\mu}{\sigma} + \beta$.
- Layer Normalization Layer
  - Normalize layers too lol
- Underfitting = high training error, overfitting = high difference in test/training error
- Capacity = ability to fit many possible functions.
  - high capacity models tends to overfit. Low tends to underfit
- Representational capacity - ability to approximate a func well. (DNNS great at)
- Effective Capacity. - what funcs the model can actually approximate
- effective = representational in convex optimization

Lec 16

Kernels are good. Maps input to higher dimension, but we only care about pairwise difference. So kernels compute pairwise differences in this higher dimension, which is usually faster than outright mapping.
$K(x, x') = K(x'x)$ computes this multidimensional pairwise difference and is kernel. Very good for SVMs.
- Polynomial, Linear, RBF kernels are useful. RBF is infinite-dimensional

```
Unimportant, extraneous others
```