Today

- Second-order optimizers use the Hessian and related matrices (e.g. $\mathbf{G}$, $\mathbf{F}$) to speed up convergence

- Motivations/Interpretations
  - Minimizing quadratic approximations
  - Preconditioning
  - Invariance to reparameterization
  - Proximal optimization

- Approximating the second-order updates
  - Conjugate gradient on batches (e.g. Hessian-free optimization)
  - Parametric approximations
    - Pullback Sampling Trick
    - K-FAC
Interpretation 1: Minimizing Quadratic Approximations
Minimizing Quadratic Approximations

Recall:

- Analyzed the behavior of gradient descent on quadratic objectives, saw that it makes slower progress in directions of low curvature (Lecture 1)
- Stationary points: \( \nabla J(w) = 0 \) (Lecture 1)
- Approximating a twice differentiable cost function using its second-order Taylor approximation (Lecture 2)

\[
J_{\text{quad}}(w) = J(w_0) + \nabla J(w_0)^\top (w - w_0) + \frac{1}{2} (w - w_0)^\top H(w - w_0)
\]

- Convex functions have PSD Hessians (Lecture 2)
- Convex quadratics can be minimized in closed form (Lecture 1)
Minimizing Quadratic Approximations

Newton’s method as solving a nonlinear equation:

- Stationary points: $\nabla J(w) = 0$
- First-order Taylor approximation to the gradient:
  \[
  \nabla J(w) \approx \nabla J(w_0) + H(w - w_0)
  \]
- Setting this to zero:
  \[
  w = w_0 - H^{-1}\nabla J(w_0)
  \]
- The Newton-Raphson method, or Newton’s method, applies this update repeatedly.
Minimizing Quadratic Approximations

Newton’s method as minimizing quadratic approximations:

- Second-order Taylor approximation to the cost:
  \[ J_{\text{quad}}(w) = J(w_0) + \nabla J(w_0)^\top (w - w_0) + \frac{1}{2}(w - w_0)^\top H(w - w_0) \]

- If \( J \) is strictly convex \( H \succ 0 \), this has a unique optimum (Lecture 1):
  \[ w = \arg \min_w J_{\text{quad}}(w) = w_0 - H^{-1} \nabla J(w_0) \]

- Newton’s method repeatedly minimizes the second-order Taylor approximation.

- This interpretation highlights that it may be useful to minimize the quadratic only approximately.
Minimizing Quadratic Approximations

- What if $J$ isn’t convex?
  - Second-order Taylor approximation may be unbounded below
  - Newton’s method just searches for stationary points (which may be saddle points)
- If we replace $H$ with a positive definite matrix $C$, i.e. compute $\Delta w = -C^{-1}\nabla J(w)$, then we are at least guaranteed to get a descent direction, i.e. a direction $\Delta w$ such that $\nabla J(w)^\top \Delta w < 0$.
- **Proof:**
  - $C^{-1} \succ 0 \iff C \succ 0$
  - $\nabla J(w)^\top \Delta w = -\nabla J(w)^\top C^{-1} \nabla J(w) < 0$ by definition of PD
- Therefore, in deep learning, we typically replace $H$ with $G$ (generalized Gauss-Newton algorithm) or $F$ (natural gradient descent).
Minimizing Quadratic Approximations: Damping

- **A problem:** if $J$ is convex but not strictly convex, then $H$ could be singular. In general, $G$ and $F$ could be singular as well.
- Even for strictly convex problems, the “vanilla” version of Newton’s method isn’t guaranteed to converge efficiently, or even reduce the cost function in each iteration.

\[ J(x) = \sum_j [\log(1 + e^{x_j}) + \log(1 + e^{-x_j})], \text{ based on logistic regression} \]
A solution: dampen the update by adding a Euclidean proximity term penalizing the distance from the current iterate (Lecture 3)

\[
\begin{align*}
\mathbf{w}^{(k+1)} &= \arg \min_{\mathbf{w}} \mathcal{J}_{\text{quad}}(\mathbf{w}) + \frac{\eta}{2} \| \mathbf{w} - \mathbf{w}^{(k)} \| ^2 \\
&= \arg \min_{\mathbf{w}} \nabla \mathcal{J}(\mathbf{w}^{(k)})^\top \mathbf{w} + \frac{1}{2} (\mathbf{w} - \mathbf{w}^{(k)})^\top (\mathbf{H} + \eta \mathbf{I})(\mathbf{w} - \mathbf{w}^{(k)}) \\
&= \mathbf{w}^{(k)} - (\mathbf{H} + \eta \mathbf{I})^{-1} \nabla \mathcal{J}(\mathbf{w}^{(k)}),
\end{align*}
\]

Here, \( \eta > 0 \) is a hyperparameter called the damping parameter.
Minimizing Quadratic Approximations: Damping

- Note that $H$, $H^{-1}$, and $(H + \eta I)^{-1}$ are all codiagonalizable (i.e. they share the same eigenvectors).
- Suppose the eigenvalues of $H$ are $\{\nu_j\}_{j=1}^D$. Since $H$ is PSD, $\nu_j \geq 0$ for all $j$.
  - The eigenvalues of $H^{-1}$ are $\{\nu_j^{-1}\}_{j=1}^D$ (assuming $H^{-1}$ exists).
  - The eigenvalues of $(H + \eta I)^{-1}$ are $\{(\nu_j + \eta)^{-1}\}_{j=1}^D$.
    - They are positive, so $(H + \eta I)^{-1}$ is positive definite, and therefore we get a descent direction.
    - They are bounded above by $\eta^{-1}$, so damping prevents the algorithm from taking extremely large steps when the curvature is close to 0.
- The damped update behaves like the undamped update in high curvature directions ($\nu_j \gg \eta$), and like gradient descent in low curvature directions ($\nu_j \ll \eta$).
Interpretation 2: Preconditioning
Preconditioning

- Recall: convergence rate of gradient descent for quadratics is determined by the condition number
  - The condition number itself isn’t defined for neural nets, since the Hessian is usually singular (more on this later)
  - But we’d still like the curvature to be reasonably well matched in all the directions that are “important” for learning
- Optimizers which compute \( w' = w - \alpha C^{-1} \nabla J(w) \) can be viewed as **preconditioned** gradient descent: implicitly doing GD in a space which is better conditioned
Preconditioning

- Consider the affine reparameterization

\[ w = \mathcal{T}(\tilde{w}) = R\tilde{w} + b, \]

where \( R \) is an invertible square matrix (not necessarily symmetric), and \( b \) is a vector.

- Inverse transformation:

\[ \tilde{w} = R^{-1}(w - b) \]

- Performing GD in the transformed space (analogous to Lecture 1):

\[
\begin{align*}
    w^{(k+1)} &= \mathcal{T}(\tilde{w}^{(k)} - \alpha \nabla \tilde{J}(\tilde{w}^{(k)})) \\
    &= \mathcal{T}(\tilde{w}^{(k)} - \alpha R^\top \nabla J(w^{(k)})) \\
    &= w^{(k)} - \alpha RR^\top \nabla J(w^{(k)}).
\end{align*}
\]

- We can get the same effect by just multiplying by \( RR^\top \) and never explicitly applying the transformation.
Preconditioning

- Turning this around: multiplying the gradient by $C^{-1}$ is equivalent to applying a transformation $R$ such that $RR^\top = C^{-1}$
  - E.g., Cholesky factorization
  - E.g., matrix square root $C^{-1/2} = QD^{-1/2}Q^\top$

- Hessian in the transformed space:
  \[
  \tilde{H} = \nabla^2 \tilde{J}(\tilde{w}) = R^\top HR
  \]

- **Corollary:** Newton’s method implicitly transforms to a space where $\tilde{H} = H^{-1/2}HH^{-1/2} = I$

- Even relatively inaccurate approximations to $H$ (e.g. diagonal) can improve the conditioning considerably

- Preconditioning is used in a lot of settings beyond optimization (e.g. solving linear systems)
Interpretation 3: Invariance
We already motivated the usefulness of invariance to reparameterizations in Chapter 3.

It can be shown that Newton-Raphson, Gauss-Newton, and natural gradient descent are all invariant to affine transformations of the parameter space (see NNTD readings).

- Intuition: if you stretch the parameter space, then the quadratic approximation gets stretched the same way as the actual cost function.

- Note: invariance only holds exactly for the undamped algorithms.
  - Damping uses a Euclidean proximity term, which depends on the coordinate system.
  - In machine learning, we don’t necessarily want full invariance, since the curvature can contain useful information about signal vs. noise.
Interpretation 4: Proximal Optimization
Recall “gradient descent on the outputs” (Lecture 3)

Roughly speaking, we can think of each update of a stochastic optimization algorithm as trading off 3 factors:

1. **Loss on the current batch.**
2. **Function space distance (FSD).** Average change to the network’s outputs. (Not necessarily a true distance metric.)
   - Prevents large steps in high-sensitivity directions (≈ high-curvature directions)
   - Saves the network from forgetting what it previously learned
3. **Weight space distance (WSD).** Typically (squared) Euclidean distance.
   - Prevents extremely large steps (damping)
   - Keeps the update within a region where the second-order approximations are accurate
   - Improves generalization by providing an inductive bias (coming up in Lecture 6)
   - Surprisingly useful for neural net training!
Generic proximal objective:

\[
\mathbf{w}^{(k+1)} \leftarrow \arg\min_{\mathbf{w}} \frac{1}{|\mathcal{B}^{(k+1)}|} \sum_{i \in \mathcal{B}^{(k+1)}} \mathcal{L}(f(x^{(i)}, \mathbf{w}), t^{(i)}) + \lambda_{\text{FSD}} \mathbb{E}_x[\rho(f(x, \mathbf{w}), f(x, \mathbf{w}^{(k)}))] + \lambda_{\text{WSD}} \frac{1}{2} \|\mathbf{w} - \mathbf{w}^{(k)}\|^2
\]

- **SGD**: linear approximation to loss, no FSD term
- **Natural gradient descent**: linear approximation to loss, quadratic approximation to FSD, WSD term = damping
Examples of proximal updates for a neural net regression problem:
Computing Second-Order and/or Proximal Updates
The matrices $H$, $G$, $F$, etc. are very large
- Small fully connected layer with 1000 inputs and 1000 outputs: 1 million parameters
- $H/G/F$ are $1 \text{ million} \times 1 \text{ million}$

How to compute $-C^{-1}\nabla J(w)$, where $C$ is one of these matrices?
- Exact inversion is hopeless
- Gradient descent on $\frac{1}{2}v^\top Cv^\top + \nabla J(w)^\top v$?
- Conjugate gradient?
- Or forget the Taylor approximation, and just do gradient descent on the proximal objective?
Consider the proximal objective, approximating FSD with the current batch:

$$Q(w) = \frac{1}{|B|} \sum_{i \in B} \left[ \mathcal{L}(f(x^{(i)}, w), t^{(i)}) + \lambda_{\text{FSD}} \rho(f(x^{(i)}, w), f(x^{(i)}, w^{(k)})) \right] + \frac{\lambda_{\text{WSD}}}{2} \| w - w^{(k)} \|^2$$

Suppose we do $K$ steps of gradient descent on this objective for each batch.

Computational cost of each step:
- Forward pass to compute $\{ f(x^{(i)}, w) \}_{i \in B}$
- Backward pass to compute the gradient of the loss and FSD terms
- Each SGD step is also a forward and a backward pass. So the cost is equivalent to $K$ SGD steps
- Is this advantageous?
Is this advantageous?

- For most supervised learning, no
  - Rather do $K$ SGD steps on fresh data than $K$ on the same batch, in order to maximize data throughput
- Can be advantageous if $K$ updates on the same data are cheaper than $K$ updates on separate batches, e.g. if disk bandwidth is the bottleneck
- Can be very advantageous if data throughput isn’t limited by computation
  - In reinforcement learning, we care about sample efficiency, i.e. the number of interactions with the environment
  - **Proximal policy optimization (PPO)** is a state-of-the-art RL algorithm used in OpenAI’s DoTA2 agent
  - It optimizes a similar proximal objective with GD (plus a few more tricks)
Gradient descent on the quadratic approximation?

$$\min_v \frac{1}{2} v^\top C v^\top + \nabla J(w)^\top v$$

Computational cost:
- Compute $\nabla J(w)$ once
- Compute an MVP $Cv$ for each subsequent step (cost $\geq 1$ gradient step on the loss and/or proximal objective)

Therefore, no computational savings from the quadratic approximation. Might as well do gradient descent on the exact (or proximal) objective.
Computation: Hessian-Free Optimization

\[ \min \frac{1}{2} v^\top C v^\top + \nabla J(w)^\top v \]

- **Hessian-free optimization (HF)** minimizes the quadratic approximation using conjugate gradient
  - Recall: CG achieves the minimum quadratic cost achievable with a given number of MVPs
  - \( \mathcal{O}(\kappa^{1/2}) \) complexity, compared with \( \mathcal{O}(\kappa) \) for gradient descent
- **Pro:** If the cost function is very ill-conditioned, \( K \) iterations of CG might make much more progress than \( K \) SGD steps
- **Con:** Each training example takes \( K \) times longer to process, so much lower data throughput
  - Note that \( K = 1 \) is equivalent to GD with automatic step size selection, so we only get a convergence benefit for larger \( K \)
- In practice, works very well for 2010-era deep networks, not so favorable for modern architectures (more on this in Lecture 7)
8. Discussion of results and implications

The most important implication of our results is that learning in deep models can be achieved effectively and efficiently by a completely general optimizer without any need for pre-training. This opens the door to examining a diverse range of deep or otherwise difficult-to-optimize architectures for which there are no effective pre-training methods, such as asymmetric auto-encoders, or recurrent neural nets.
Pullback Sampling Trick
Pullback Sampling Trick

- Limitations of MVP-based methods
  - Requirement of doing many MVPs hurts data throughput
  - Approximate curvature or FSD using a single batch, which may be inaccurate

- A more recent approach: fit tractable parametric approximations to $G$ or $F$

- **Pullback Sampling Trick (PST):** sample vectors $ Dw $, called pseudo-gradients, whose covariance is $ G $ (or $ F $)
  - Then we can fit a tractable probabilistic model to approximate this covariance
Pullback Sampling Trick

Using the PST to estimate $\mathbf{F}$ for a linear regression model

Recall:

$\mathbf{F} = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \left[ \mathcal{D} \mathbf{w} \mathcal{D} \mathbf{w}^\top \right]$

$\mathbf{F} = \mathbf{G}$ for exponential family NLL (squared error = Gaussian NLL)
Pullback Sampling Trick

\[ G = \mathbb{E}_x[J_{zw}^\top G_z J_{zw}] \]

- **Pullback Sampling Trick (PST):** sample pseudo-gradients \( Dw \) whose covariance is \( G \), and approximate the covariance
  - Sample \( x \) from the data distribution
  - Compute \( z = f(x, w) \)
  - Sample a random vector \( Dz \) whose covariance is \( G_z \)
  - Pull it back to weight space using a JVP (i.e. backprop):
    \[ Dw = J_{zw}^\top Dz \]

- The resulting random vector \( Dw \) has covariance
  \[ \mathbb{E}[J_{zw}^\top G_z J_{zw}] = G_w. \]
Pullback Sampling Trick

- The simplest structure we can impose on $\mathbf{G}$ is diagonal
  - Equivalent to approximating the entries of $\mathbf{Dw}$ as uncorrelated (or independent)
  - To compute $\hat{\mathbf{G}}^{-1}$, just invert the diagonal entries
- Estimate from a finite set of samples:

  \[
  \hat{\mathbf{G}}_{ii} = \frac{1}{S} \sum_{s=1}^{S} \mathbf{D}w_i^2
  \]

- In practice, often use an exponential moving average (EMA):

  \[
  \hat{\mathbf{G}}_{ii}^{(k+1)} \leftarrow \eta \hat{\mathbf{G}}_{ii}^{(k)} + (1 - \eta)[\mathbf{D}w_i^{(k)}]^2
  \]

- $\eta$ is a hyperparameter (good default: 0.95)
- $1/(1 - \eta)$ is the timescale of the EMA
Kronecker-Factored Approximate Curvature
Can we do better than a diagonal approximation?

Probabilistic graphical models (PGMs) give us a powerful set of techniques for efficiently approximating high-dimensional probability distributions.

Kronecker-Factored Approximate Curvature (K-FAC) fits a structured probabilistic model to the gradient computations in order to cheaply approximate the Gauss-Newton update or natural gradient.
K-FAC: Kronecker Product

- The vectorization operator $\text{vec}(A)$ stacks the columns of a matrix $A$ into a vector.
- Kronecker product:

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{pmatrix}$$
The Kronecker product is useful since it lets us express matrix multiplication as a linear operator:

\[ \text{vec}(AXB) = (B^\top \otimes A) \text{vec}(X) \]

Proof-by-picture of a special case, \( \text{vec}(AX) = (I \otimes A) \text{vec}(X) \):

\[
\begin{pmatrix}
\text{vec}(AX)
\end{pmatrix}
\begin{pmatrix}
\cdot \\
\cdot \\
\cdot \\
\cdot 
\end{pmatrix}
\]
K-FAC: Kronecker Product

Some properties of the Kronecker product:

- **Matrix multiplication:**
  \[(A \otimes B)(C \otimes D) = AC \otimes BD\]

- **Matrix transpose:**
  \[(A \otimes B)^\top = A^\top \otimes B^\top\]

- **Matrix inversion:**
  \[(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}\]

- **Vector outer products (u and v are column vectors):**
  \[\text{vec}(uv^\top) = v \otimes u\]
K-FAC: Kronecker Product

- If $Q_1$ and $Q_2$ are orthogonal, then so is $Q_1 \otimes Q_2$.
- If $D_1$ and $D_2$ are diagonal, then so is $D_1 \otimes D_2$.
- If $A$ and $B$ are symmetric, then so is $A \otimes B$.
- Spectral decomposition for symmetric $A = Q_A D_A Q_A^\top$ and $B = Q_B D_B Q_B^\top$

$$A \otimes B = (Q_A \otimes Q_B)(D_A \otimes D_B)(Q_A^\top \otimes Q_B^\top)$$

- Therefore, if the eigenvalues of $A$ are $\lambda_i$ and the eigenvalues of $B$ are $\nu_j$, then the eigenvalues of $A \otimes B$ are the products $\lambda_i \nu_j$.
- If the corresponding eigenvectors of $A$ are $r_i$ and for $B$ are $s_j$, then the eigenvectors of $A \otimes B$ are $r_i \otimes s_j$.
- Therefore if $A$ and $B$ are positive (semi)definite, then so is $A \otimes B$. 
K-FAC: Modeling the Pseudo-Gradients

- Computations in each layer of an MLP:
  \[ s_\ell = \bar{W}_\ell \bar{a}_{\ell-1} \]
  \[ a_\ell = \phi_\ell(s_\ell) \]

- Backprop computations in each layer:
  \[ D a_\ell = W_{\ell}^\top D s_{\ell+1} \]
  \[ D s_\ell = D a_\ell \odot \phi'_\ell(s_\ell) \]
  \[ D \bar{W}_\ell = D s_\ell \bar{a}_{\ell-1}^\top \]
**Approximation 1:** different layers are independent

This makes $G$ into a block diagonal matrix, with one block per layer of the network.

\[
G_{\ell\ell} = \mathbb{E}[\text{vec}(\mathcal{D}W_\ell) \text{vec}(\mathcal{D}W_\ell)^\top]
\]
\[
= \mathbb{E}[\text{vec}(\mathcal{D}s_\ell \bar{a}_\ell^{\top}) \text{vec}(\mathcal{D}s_\ell \bar{a}_\ell^{\top})^\top]
\]
\[
= \mathbb{E}[(\bar{a}_{\ell-1} \otimes \mathcal{D}s_\ell)(\bar{a}_{\ell-1} \otimes \mathcal{D}s_\ell)^\top]
\]
\[
= \mathbb{E}[(\bar{a}_{\ell-1}^{\top}) \bar{a}_{\ell-1}^{\top} \otimes \mathcal{D}s_\ell \mathcal{D}s_\ell^\top]
\]

- The blocks are still too large!
Approximation 2: $\bar{a}_{\ell-1}$ is independent of $D_s\ell$

Then we can push the expectation inwards and get a Kronecker product:

$$\hat{G}_{\ell\ell} = E[\bar{a}_{\ell-1}\bar{a}_\ell^\top] \otimes E[D_s\ell D_s\ell^\top]$$

$$= A_{\ell-1} \otimes S_\ell,$$

where $A_\ell$ and $S_\ell$ denote the following covariance matrices:

$$A_\ell = E[\bar{a}_\ell\bar{a}_\ell^\top]$$

$$= \begin{pmatrix} E[a_\ell a_\ell^\top] & E[a_\ell] \\ E[a_\ell^\top] & 1 \end{pmatrix}$$

$$S_\ell = E[D_s\ell D_s\ell^\top]$$
K-FAC: Compact Representation

How large is the representation?

- Assume 3 layers with 1000 units per layer
- Full matrix $\mathbf{G}$:
  \[(3 \times 1000^2)^2 = 9 \text{ trillion}\]
- Block diagonal:
  \[3 \times (1000^2)^2 = 3 \text{ trillion}\]
- Kronecker-factored ($\hat{\mathbf{G}}_{\ell\ell} = \mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell$):
  \[3 \times (1000^2 + 1000^2) = 6 \text{ million}\]
Efficiently solving the linear system:

\[ \hat{G}_{\ell \ell}^{-1} v_\ell = (A_{\ell-1} \otimes S_{\ell})^{-1} \text{vec}(\bar{V}_\ell) \]

\[ = (A_{\ell-1}^{-1} \otimes S_{\ell}^{-1}) \text{vec}(\bar{V}_\ell) \]

\[ = \text{vec}(S_{\ell}^{-1} \bar{V}_\ell A_{\ell-1}^{-1}). \]

This only requires computations with matrices that are approximately the same size as the weights.

**Note:** this update rule needs to be modified to approximate the damped update, \((\hat{G}_{\ell \ell} + \eta I)^{-1} v_\ell\). Details in the readings.
Estimate the Kronecker factors \( \{A_\ell\} \) and \( \{S_\ell\} \) with exponential moving averages

\[
\hat{A}_\ell \leftarrow \eta \hat{A}_\ell + \frac{1 - \eta}{B} Y_\ell^\top Y_\ell
\]
\[
\hat{S}_\ell \leftarrow \eta \hat{S}_\ell + \frac{1 - \eta}{B} DZ_\ell^\top DZ_\ell,
\]

where \( Y \) and \( Z \) denote the matrices of activations and pre-activations for a batch.
K-FAC: Odds and Ends

Some things we left out:

- Adding momentum and iterate averaging (straightforward)
- Using exact MVPs on the current batch to choose step sizes automatically
- Automatically adapting damping hyperparameters
- More accurate approximations than layerwise independence
- Extensions to other architectures (conv nets, RNNs, etc.)
- Distributed implementation
K-FAC: Results

Logistic autoencoder

![Graph showing error vs iterations for Logistic autoencoder with different K-FAC strategies.]

RNN language model

![Graph showing validation perplexity vs seconds for RNN language model with different optimization methods.]

ImageNet classifier CNN

![Graph showing error vs updates for ImageNet classifier CNN with different K-FAC strategies.]

FermiNET
(Schrödinger Equation)

![Graph showing energy vs iterations for FermiNET with K-FAC and Adam.]

- Results are sometimes amazing, sometimes meh.