CSC 2541: Neural Net Training Dynamics

Lecture 4 - Second-Order Optimization

Roger Grosse

University of Toronto, Winter 2021

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: $\nabla \mathcal{J}(\mathbf{w}) = \mathbf{0}$ (Lecture 1)
- Approximating a twice differentiable cost function using its second-order Taylor approximation (Lecture 2)

$$\mathcal{J}_{\text{quad}}(\mathbf{w}) = \mathcal{J}(\mathbf{w}_0) + \nabla \mathcal{J}(\mathbf{w}_0)^\top (\mathbf{w} - \mathbf{w}_0) + \tfrac{1}{2}(\mathbf{w} - \mathbf{w}_0)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_0)$$

- 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:

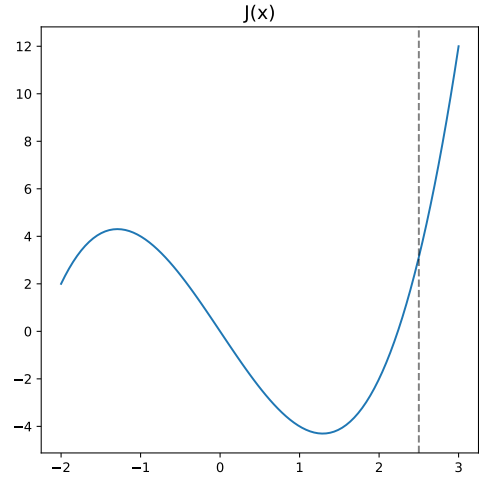
- Stationary points: $\nabla \mathcal{J}(\mathbf{w}) = \mathbf{0}$
- First-order Taylor approximation to the gradient:

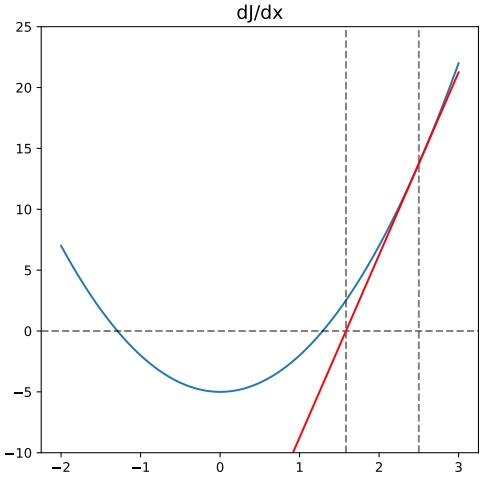
$$\nabla \mathcal{J}(\mathbf{w}) \approx \nabla \mathcal{J}(\mathbf{w}_0) + \mathbf{H}(\mathbf{w} - \mathbf{w}_0)$$

- Setting this to zero:

$$\mathbf{w} = \mathbf{w}_0 - \mathbf{H}^{-1} \nabla \mathcal{J}(\mathbf{w}_0)$$

- The Newton-Raphson method, or Newton's method, applies this update repeatedly.





Minimizing Quadratic Approximations

Newton's method as minimizing quadratic approximations:

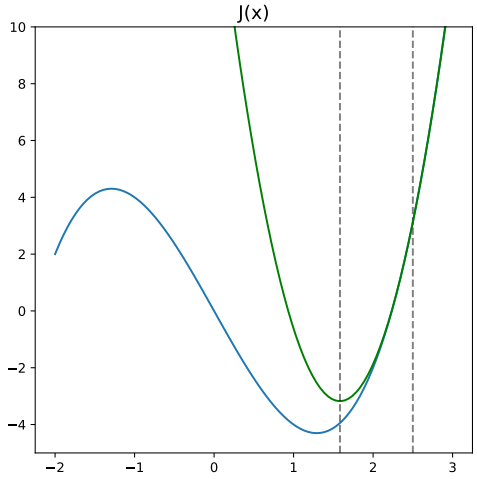
- Second-order Taylor approximation to the cost:

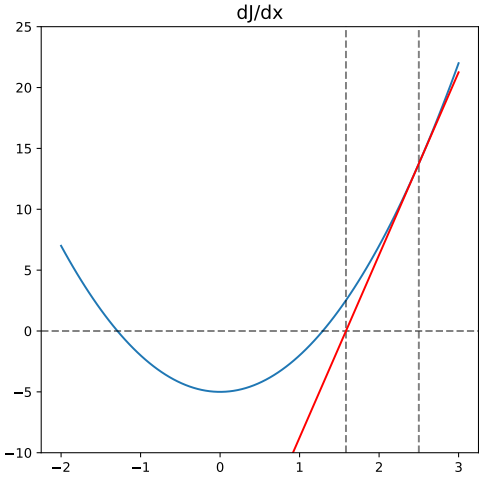
$$\mathcal{J}_{\text{quad}}(\mathbf{w}) = \mathcal{J}(\mathbf{w}_0) + \nabla\mathcal{J}(\mathbf{w}_0)^\top(\mathbf{w} - \mathbf{w}_0) + \\ + \tfrac{1}{2}(\mathbf{w} - \mathbf{w}_0)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}_0)$$

- If $\mathcal{J}$ is strictly convex $\mathbf{H} \succ \mathbf{0}$, this has a unique optimum (Lecture 1):

$$\mathbf{w} = \arg\min_{\mathbf{w}} \mathcal{J}_{\text{quad}}(\mathbf{w}) = \mathbf{w}_0 - \mathbf{H}^{-1}\nabla\mathcal{J}(\mathbf{w}_0)$$

- Newton's method repeatedly minimizes the second-order Taylor approximation.
- This interpretation highlights that it may be useful to minimize the quadratic only approximately.



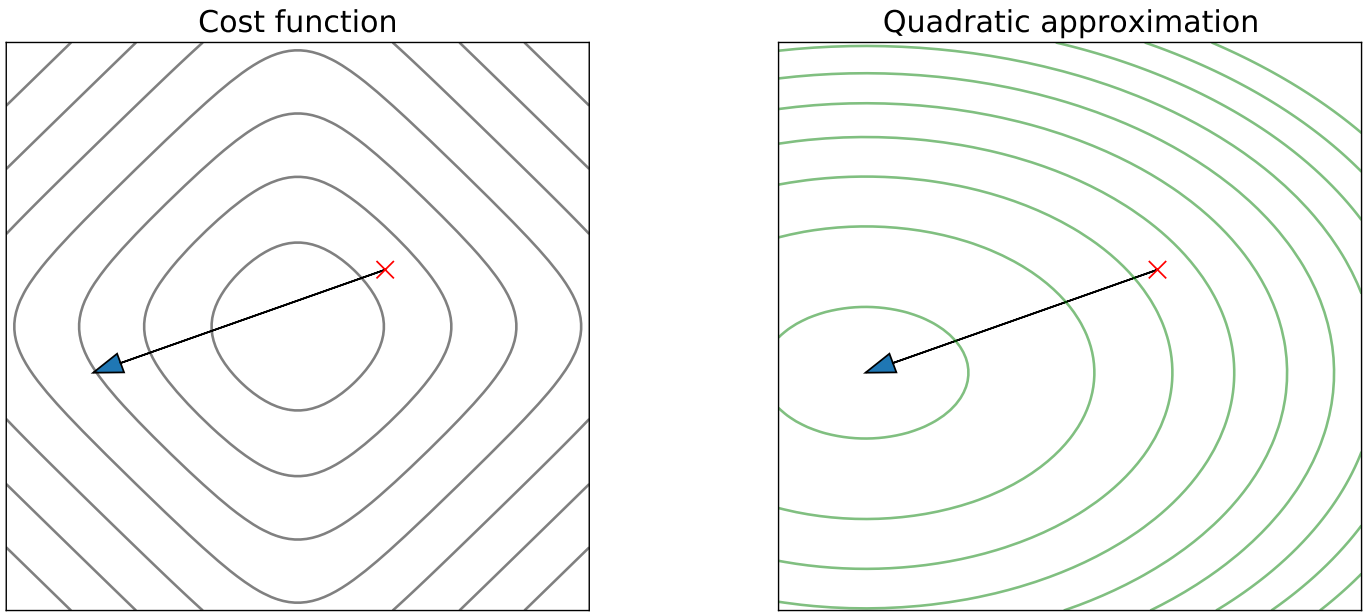


Minimizing Quadratic Approximations

- What if $\mathcal{J}$ 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 $\mathbf{H}$ with a positive definite matrix $\mathbf{C}$, i.e. compute $\Delta\mathbf{w} = -\mathbf{C}^{-1}\nabla\mathcal{J}(\mathbf{w})$, then we are at least guaranteed to get a descent direction, i.e. a direction $\Delta\mathbf{w}$ such that $\nabla\mathcal{J}(\mathbf{w})^\top\Delta\mathbf{w} < 0$.
- **Proof:**
  - $\mathbf{C}^{-1} \succ \mathbf{0} \iff \mathbf{C} \succ \mathbf{0}$
  - $\nabla\mathcal{J}(\mathbf{w})^\top\Delta\mathbf{w} = -\nabla\mathcal{J}(\mathbf{w})^\top\mathbf{C}^{-1}\nabla\mathcal{J}(\mathbf{w}) < 0$ by definition of PD
- Therefore, in deep learning, we typically replace $\mathbf{H}$ with $\mathbf{G}$ (generalized Gauss-Newton algorithm) or $\mathbf{F}$ (natural gradient descent).

Minimizing Quadratic Approximations: Damping

- **A problem:** if $\mathcal{J}$ is convex but not strictly convex, then $\mathbf{H}$ could be singular. In general, $\mathbf{G}$ and $\mathbf{F}$ 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



$\mathcal{J}(\mathbf{x}) = \sum_j [\log(1 + e^{x_j}) + \log(1 + e^{-x_j})]$, based on logistic regression

Minimizing Quadratic Approximations: Damping

- **A solution:** dampen the update by adding a Euclidean proximity term penalizing the distance from the current iterate (Lecture 3)

$$\begin{aligned}\mathbf{w}^{(k+1)} &= \arg\min_{\mathbf{w}} \mathcal{J}_{\text{quad}}(\mathbf{w}) + \tfrac{\eta}{2}\|\mathbf{w} - \mathbf{w}^{(k)}\|^2 \\ &= \arg\min_{\mathbf{w}} \nabla\mathcal{J}(\mathbf{w}^{(k)})^\top \mathbf{w} + \tfrac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top (\mathbf{H} + \eta\mathbf{I})(\mathbf{w} - \mathbf{w}^{(k)}) \\ &= \mathbf{w}^{(k)} - (\mathbf{H} + \eta\mathbf{I})^{-1}\nabla\mathcal{J}(\mathbf{w}^{(k)}),\end{aligned}$$

- Here, $\eta > 0$ is a hyperparameter called the damping parameter

Minimizing Quadratic Approximations: Damping

- Note that $\mathbf{H}$, $\mathbf{H}^{-1}$, and $(\mathbf{H}+\eta\mathbf{I})^{-1}$ are all codiagonalizable (i.e. they share the same eigenvectors)
- Suppose the eigenvalues of $\mathbf{H}$ are $\{\nu_j\}_{j=1}^D$. Since $\mathbf{H}$ is PSD, $\nu_j \geq 0$ for all j.
  - The eigenvalues of $\mathbf{H}^{-1}$ are $\{\nu_j^{-1}\}_{j=1}^D$ (assuming $\mathbf{H}^{-1}$ exists).
  - The eigenvalues of $(\mathbf{H}+\eta\mathbf{I})^{-1}$ are $\{(\nu_j+\eta)^{-1}\}_{j=1}^D$.
    - They are positive, so $(\mathbf{H}+\eta\mathbf{I})^{-1}$ is positive definite, and therefore we get a descent direction.
    - They are bounded above by η^{-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 $\mathbf{w}' = \mathbf{w} - \alpha \mathbf{C}^{-1} \nabla \mathcal{J}(\mathbf{w})$ can be viewed as preconditioned gradient descent: implicitly doing GD in a space which is better conditioned

Preconditioning

- Consider the affine reparameterization

$$\mathbf{w} = \mathcal{T}(\tilde{\mathbf{w}}) = \mathbf{R}\tilde{\mathbf{w}} + \mathbf{b},$$

  where $\mathbf{R}$ is an invertible square matrix (not necessarily symmetric), and $\mathbf{b}$ is a vector
- Inverse transformation:

$$\tilde{\mathbf{w}} = \mathbf{R}^{-1}(\mathbf{w} - \mathbf{b})$$

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

$$\begin{aligned}
\mathbf{w}^{(k+1)} &= \mathcal{T}(\tilde{\mathbf{w}}^{(k)} - \alpha \nabla \tilde{\mathcal{J}}(\tilde{\mathbf{w}}^{(k)})) \\
&= \mathcal{T}(\tilde{\mathbf{w}}^{(k)} - \alpha \mathbf{R}^\top \nabla \mathcal{J}(\mathbf{w}^{(k)})) \\
&= \mathbf{w}^{(k)} - \alpha \mathbf{R}\mathbf{R}^\top \nabla \mathcal{J}(\mathbf{w}^{(k)}).
\end{aligned}$$

- We can get the same effect by just multiplying by $\mathbf{R}\mathbf{R}^\top$ and never explicitly applying the transformation

Preconditioning

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

$$\tilde{\mathbf{H}} = \nabla^2 \tilde{\mathcal{J}}(\tilde{\mathbf{w}}) = \mathbf{R}^\top \mathbf{H} \mathbf{R}$$

- **Corollary:** Newton's method implicitly transforms to a space where $\tilde{\mathbf{H}} = \mathbf{H}^{-1/2}\mathbf{H}\mathbf{H}^{-1/2} = \mathbf{I}$
- Even relatively inaccurate approximations to $\mathbf{H}$ (e.g. diagonal) can improve the conditioning considerably
- Preconditioning is used in a lot of settings beyond optimziation (e.g. solving linear systems)

Interpretation 3: Invariance

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

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 ($\approx$ 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!

Proximal Optimization

- **Generic proximal objective:**

$$\mathbf{w}^{(k+1)} \leftarrow \arg\min_{\mathbf{w}} \frac{1}{|\mathcal{B}^{(k+1)}|} \sum_{i \in \mathcal{B}^{(k+1)}} \mathcal{L}(f(\mathbf{x}^{(i)}, \mathbf{w}), \mathbf{t}^{(i)}) + \quad \text{(loss)}$$

$$+ \ \lambda_{\text{FSD}} \mathbb{E}_{\mathbf{x}}[\rho(f(\mathbf{x}, \mathbf{w}), f(\mathbf{x}, \mathbf{w}^{(k)}))] + \quad \text{(FSD)}$$

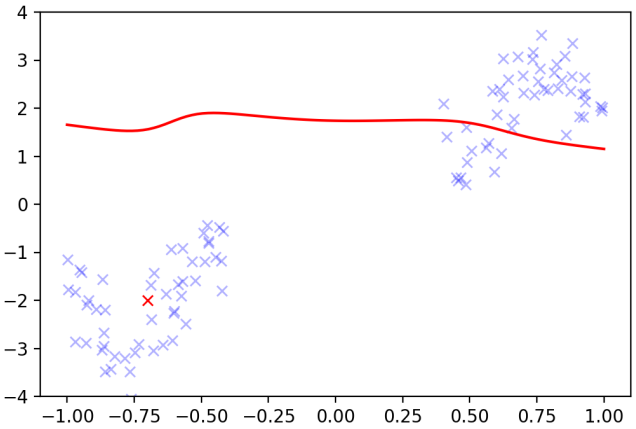
$$+ \ \frac{\lambda_{\text{WSD}}}{2} \|\mathbf{w} - \mathbf{w}^{(k)}\|^2 \quad \text{(WSD)}$$

- **SGD:** linear approximation to loss, no FSD term
- **Natural gradient descent:** linear approximation to loss, quadratic approximation to FSD, WSD term = damping

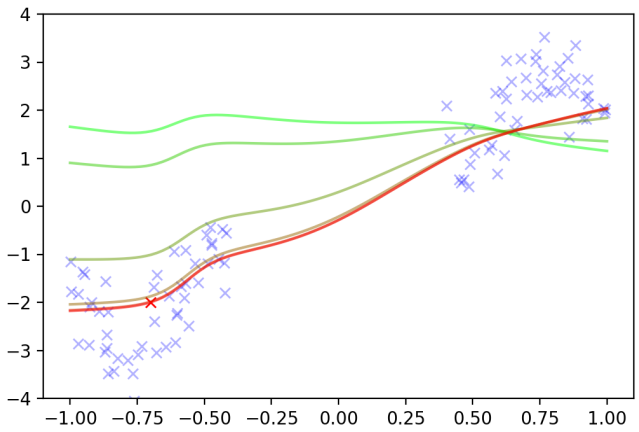
Proximal Optimization

Examples of proximal updates for a neural net regression problem:

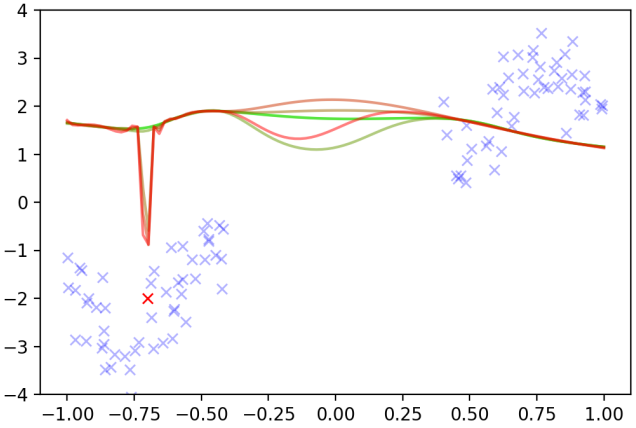
Previous iterate



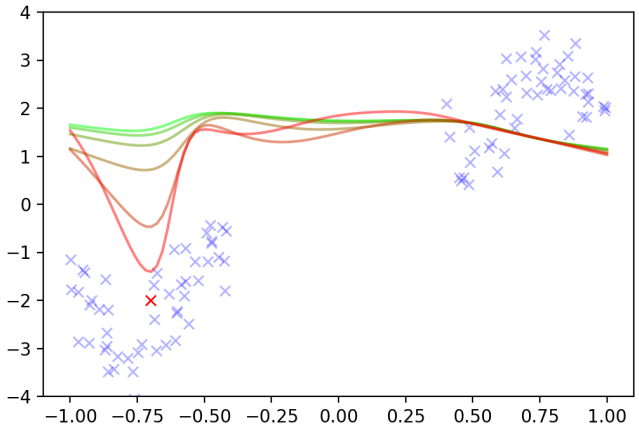
Loss + WSD



Loss + FSD



Loss + WSD + FSD



Computing Second-Order and/or Proximal Updates

Computation

- The matrices $\mathbf{H}$, $\mathbf{G}$, $\mathbf{F}$, etc. are very large
  - Small fully connected layer with 1000 inputs and 1000 outputs: 1 million parameters
  - $\mathbf{H}/\mathbf{G}/\mathbf{F}$ are 1 million $\times$ 1 million
- How to compute $-\mathbf{C}^{-1}\nabla\mathcal{J}(\mathbf{w})$, where $\mathbf{C}$ is one of these matrices?
  - Exact inversion is hopeless
  - Gradient descent on $\frac{1}{2}\mathbf{v}^\top\mathbf{C}\mathbf{v}^\top + \nabla\mathcal{J}(\mathbf{w})^\top\mathbf{v}$?
  - Conjugate gradient?
  - Or forget the Taylor approximation, and just do gradient descent on the proximal objective?

Computation: GD on the Proximal Objective

- Consider the proximal objective, approximating FSD with the current batch:

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

- Suppose we do K steps of gradient descent on this objective for each batch.
- Computational cost of each step:
  - Forward pass to compute $\{f(\mathbf{x}^{(i)}, \mathbf{w})\}_{i \in \mathcal{B}}$
  - 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?

Computation: GD on the Proximal Objective

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)

Computation: GD on the Taylor Approximation

- Gradient descent on the quadratic approximation?

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

- Computational cost:
  - Compute $\nabla \mathcal{J}(\mathbf{w})$ once
  - Compute an MVP $\mathbf{C}\mathbf{v}$ for each subsequent step (cost ≥ 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.

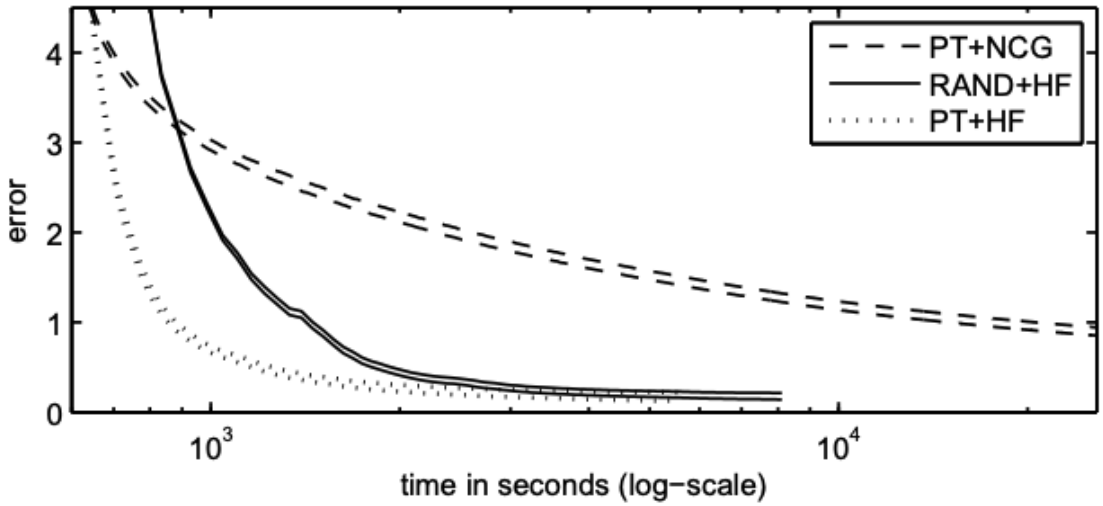
Computation: Hessian-Free Optimization

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

- 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)

Computation: Hessian-Free Optimization

- Martens (2010): training deep autoencoders without pre-training



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.

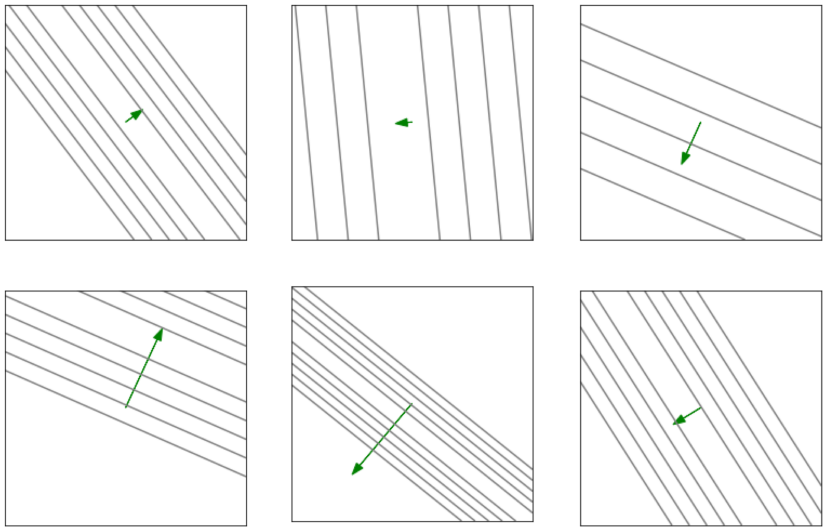
Pullback Sampling Trick

Pullback Sampling Trick

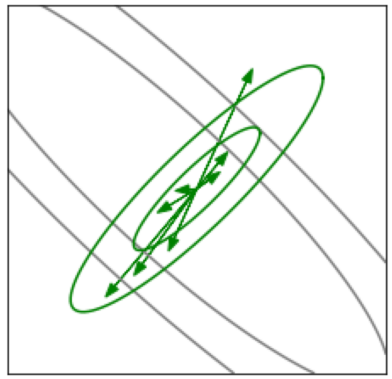
- Limitations of MVP-based methods
  - Requirement of doing many MVPs hurts data throughput
  - Approximate curvature or FSD using a single batch, which may be inaccurate
- A more recent approach: fit tractable parametric approximations to $\mathbf{G}$ or $\mathbf{F}$
- Pullback Sampling Trick (PST): sample vectors $\mathcal{D}\mathbf{w}$, called pseudo-gradients, whose covariance is $\mathbf{G}$ (or $\mathbf{F}$)
  - 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



Log-likelihood contours and gradients for data points sampled from the model's predictions



Average log-likelihood contour and distribution of gradients

Recall:

- $\mathbf{F} = \mathbb{E}_{\substack{\mathbf{x}\sim p_{\text{data}} \\ \mathbf{t}\sim r(\cdot\,|\,\mathbf{x})}} [\mathcal{D}\mathbf{w}\mathcal{D}\mathbf{w}^\top]$
- $\mathbf{F} = \mathbf{G}$ for exponential family NLL (squared error = Gaussian NLL)

Pullback Sampling Trick

$$\mathbf{G} = \mathbb{E}_{\mathbf{x}}[\mathbf{J}_{\mathbf{zw}}^{\top}\mathbf{G}_{\mathbf{z}}\mathbf{J}_{\mathbf{zw}}]$$

- Pullback Sampling Trick (PST): sample pseudo-gradients $\mathcal{D}\mathbf{w}$ whose covariance is $\mathbf{G}$, and approximate the covariance
  - Sample $\mathbf{x}$ from the data distribution
  - Compute $\mathbf{z} = f(\mathbf{x}, \mathbf{w})$
  - Sample a random vector $\mathcal{D}\mathbf{z}$ whose covariance is $\mathbf{G}_{\mathbf{z}}$
  - Pull it back to weight space using a JVP (i.e. backprop): $\mathcal{D}\mathbf{w} = \mathbf{J}_{\mathbf{zw}}^{\top}\mathcal{D}\mathbf{z}$
- The resulting random vector $\mathcal{D}\mathbf{w}$ has covariance $\mathbb{E}[\mathbf{J}_{\mathbf{zw}}^{\top}\mathbf{G}_{\mathbf{z}}\mathbf{J}_{\mathbf{zw}}] = \mathbf{G}_{\mathbf{w}}$.

Pullback Sampling Trick

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

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

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

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

  - η is a hyperparameter (good default: 0.95)
  - $1/(1-\eta)$ is the timescale of the EMA

Kronecker-Factored Approximate Curvature

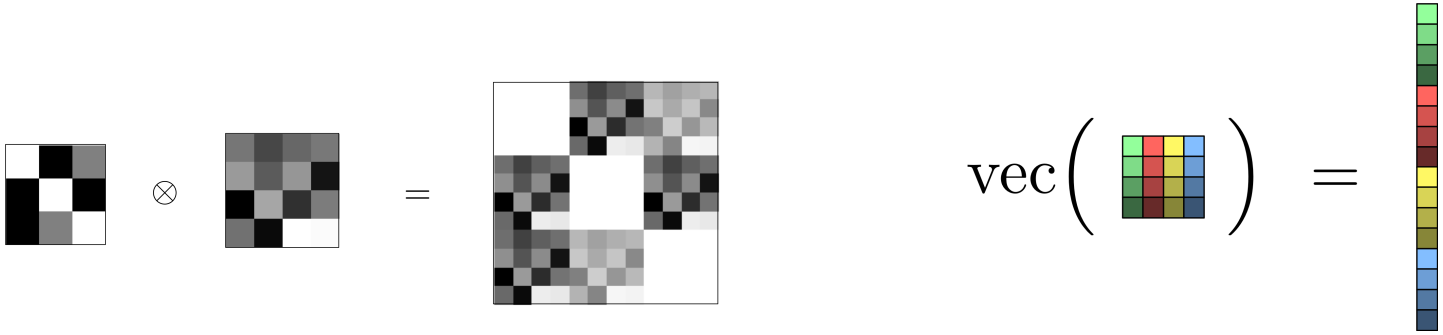
K-FAC

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

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & & a_{2n}\mathbf{B} \\ \vdots & & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{pmatrix}$$

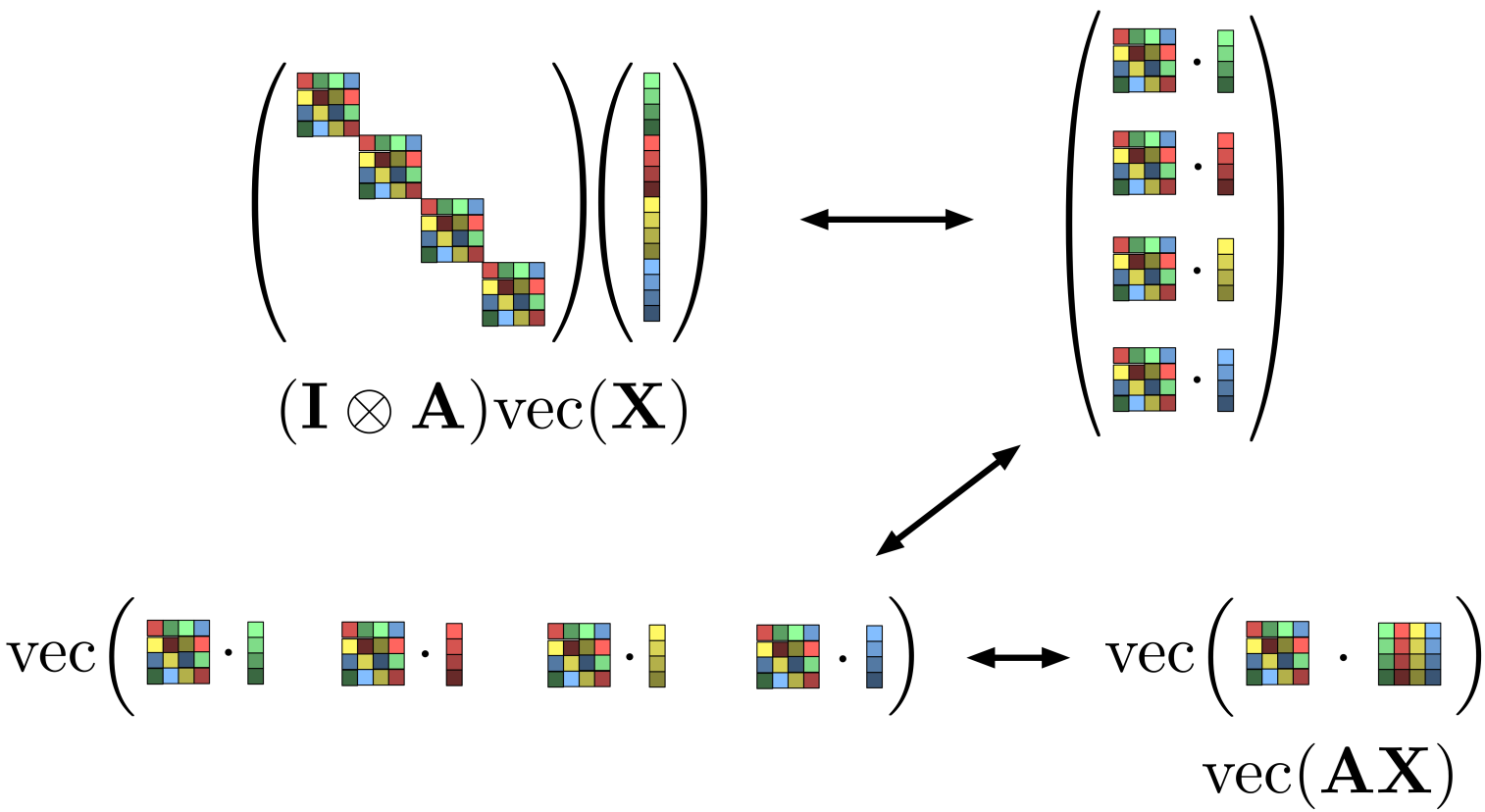


K-FAC: Kronecker Product

- The Kronecker product is useful since it lets us express matrix multiplication as a linear operator:

$$\text{vec}(\mathbf{AXB}) = (\mathbf{B}^\top \otimes \mathbf{A})\,\text{vec}(\mathbf{X})$$

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



K-FAC: Kronecker Product

Some properties of the Kronecker product:

- Matrix multiplication:

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}$$

- Matrix transpose:

$$(\mathbf{A} \otimes \mathbf{B})^\top = \mathbf{A}^\top \otimes \mathbf{B}^\top$$

- Matrix inversion:

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}$$

- Vector outer products ($\mathbf{u}$ and $\mathbf{v}$ are column vectors):

$$\text{vec}(\mathbf{uv}^\top) = \mathbf{v} \otimes \mathbf{u}$$

K-FAC: Kronecker Product

- If $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are orthogonal, then so is $\mathbf{Q}_1 \otimes \mathbf{Q}_2$.
- If $\mathbf{D}_1$ and $\mathbf{D}_2$ are diagonal, then so is $\mathbf{D}_1 \otimes \mathbf{D}_2$.
- If $\mathbf{A}$ and $\mathbf{B}$ are symmetric, then so is $\mathbf{A} \otimes \mathbf{B}$.
- Spectral decomposition for symmetric $\mathbf{A} = \mathbf{Q_A} \mathbf{D_A} \mathbf{Q_A}^\top$ and $\mathbf{B} = \mathbf{Q_B} \mathbf{D_B} \mathbf{Q_B}^\top$

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

- Therefore, if the eigenvalues of $\mathbf{A}$ are λ_i and the eigenvalues of $\mathbf{B}$ are ν_j, then the eigenvalues of $\mathbf{A} \otimes \mathbf{B}$ are the products $\lambda_i \nu_j$
- If the corresponding eigenvectors of $\mathbf{A}$ are $\mathbf{r}_i$ and for $\mathbf{B}$ are $\mathbf{s}_j$, then the eigenvectors of $\mathbf{A} \otimes \mathbf{B}$ are $\mathbf{r}_i \otimes \mathbf{s}_j$
- Therefore if $\mathbf{A}$ and $\mathbf{B}$ are positive (semi)definite, then so is $\mathbf{A} \otimes \mathbf{B}$

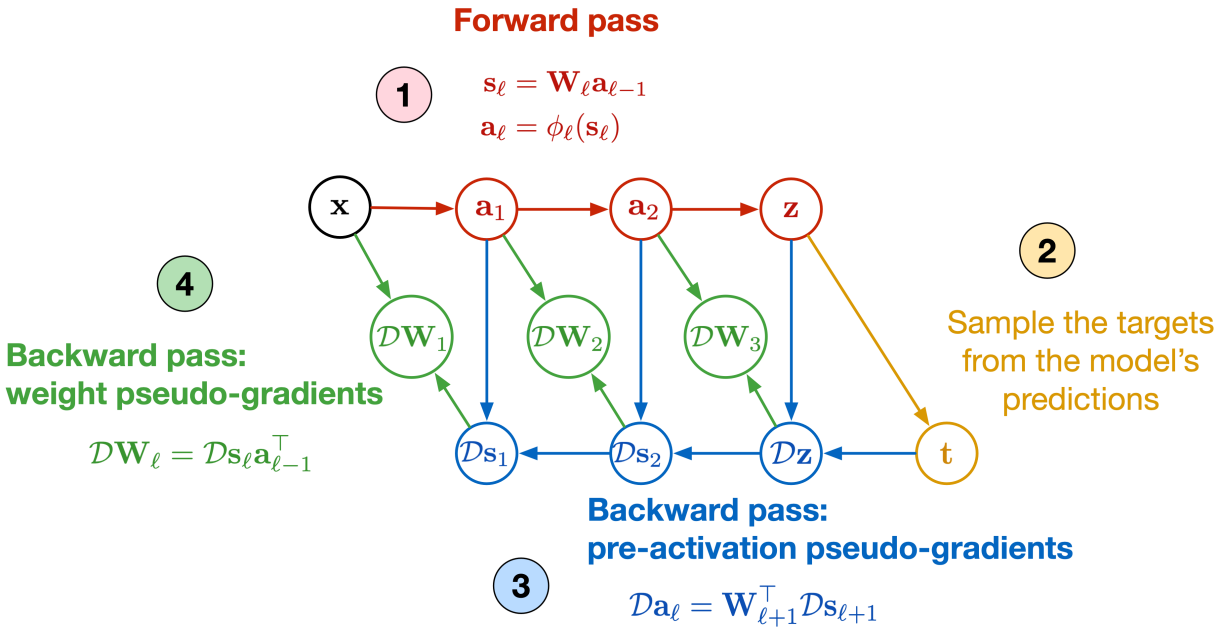
K-FAC: Modeling the Pseudo-Gradients

- Computations in each layer of an MLP:

$$\mathbf{s}_\ell = \bar{\mathbf{W}}_\ell \bar{\mathbf{a}}_{\ell-1}$$
$$\mathbf{a}_\ell = \phi_\ell(\mathbf{s}_\ell)$$

- Backprop computations in each layer:

$$\mathcal{D}\mathbf{a}_\ell = \mathbf{W}_\ell^\top \mathcal{D}\mathbf{s}_{\ell+1}$$
$$\mathcal{D}\mathbf{s}_\ell = \mathcal{D}\mathbf{a}_\ell \odot \phi_\ell'(\mathbf{s}_\ell)$$
$$\mathcal{D}\bar{\mathbf{W}}_\ell = \mathcal{D}\mathbf{s}_\ell \bar{\mathbf{a}}_{\ell-1}^\top$$



K-FAC: Modeling the Pseudo-Gradients

- **Approximation 1:** different layers are independent
- This makes $\mathbf{G}$ into a block diagonal matrix, with one block per layer of the network.

$$\begin{aligned}
\mathbf{G}_{\ell\ell} &= \mathbb{E}[\mathrm{vec}(\mathcal{D}\mathbf{W}_\ell)\,\mathrm{vec}(\mathcal{D}\mathbf{W}_\ell)^\top] \\
&= \mathbb{E}[\mathrm{vec}(\mathcal{D}\mathbf{s}_\ell\bar{\mathbf{a}}_{\ell-1}^\top)\,\mathrm{vec}(\mathcal{D}\mathbf{s}_\ell\bar{\mathbf{a}}_{\ell-1}^\top)^\top] \\
&= \mathbb{E}[(\bar{\mathbf{a}}_{\ell-1}\otimes\mathcal{D}\mathbf{s}_\ell)(\bar{\mathbf{a}}_{\ell-1}\otimes\mathcal{D}\mathbf{s}_\ell)^\top] \\
&= \mathbb{E}[\bar{\mathbf{a}}_{\ell-1}\bar{\mathbf{a}}_{\ell-1}^\top\otimes\mathcal{D}\mathbf{s}_\ell\mathcal{D}\mathbf{s}_\ell^\top]
\end{aligned}$$

- The blocks are still too large!

K-FAC: Modeling the Pseudo-Gradients

- **Approximation 2:** $\bar{\mathbf{a}}_{\ell-1}$ is independent of $\mathcal{D}\mathbf{s}_\ell$
- Then we can push the expectation inwards and get a Kronecker product:

$$\begin{aligned}\hat{\mathbf{G}}_{\ell\ell} &= \mathbb{E}[\bar{\mathbf{a}}_{\ell-1}\bar{\mathbf{a}}_{\ell-1}^\top] \otimes \mathbb{E}[\mathcal{D}\mathbf{s}_\ell \mathcal{D}\mathbf{s}_\ell^\top] \\ &= \mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell,\end{aligned}$$

where $\mathbf{A}_\ell$ and $\mathbf{S}_\ell$ denote the following covariance matrices:

$$\begin{aligned}\mathbf{A}_\ell &= \mathbb{E}[\bar{\mathbf{a}}_\ell \bar{\mathbf{a}}_\ell^\top] \\ &= \begin{pmatrix} \mathbb{E}[\mathbf{a}_\ell \mathbf{a}_\ell^\top] & \mathbb{E}[\mathbf{a}_\ell] \\ \mathbb{E}[\mathbf{a}_\ell^\top] & 1 \end{pmatrix} \\ \mathbf{S}_\ell &= \mathbb{E}[\mathcal{D}\mathbf{s}_\ell \mathcal{D}\mathbf{s}_\ell^\top]\end{aligned}$$

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 ($\hat{\mathbf{G}}_{\ell\ell} = \mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell$):

$$3 \times (1000^2 + 1000^2) = 6 \text{ million}$$

K-FAC: Efficient Computation

- Efficiently solving the linear system:

$$\begin{aligned}\hat{\mathbf{G}}_{\ell\ell}^{-1}\mathbf{v}_\ell &= (\mathbf{A}_{\ell-1} \otimes \mathbf{S}_\ell)^{-1} \operatorname{vec}(\bar{\mathbf{V}}_\ell) \\ &= (\mathbf{A}_{\ell-1}^{-1} \otimes \mathbf{S}_\ell^{-1}) \operatorname{vec}(\bar{\mathbf{V}}_\ell) \\ &= \operatorname{vec}(\mathbf{S}_\ell^{-1}\bar{\mathbf{V}}_\ell\mathbf{A}_{\ell-1}^{-1}).\end{aligned}$$

- 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, $(\hat{\mathbf{G}}_{\ell\ell} + \eta\mathbf{I})^{-1}\mathbf{v}_\ell$. Details in the readings.

K-FAC: Estimating the Covariance Matrices

- Estimate the Kronecker factors $\{\mathbf{A}_\ell\}$ and $\{\mathbf{S}_\ell\}$ with exponential moving averages

$$\hat{\mathbf{A}}_\ell \leftarrow \eta \hat{\mathbf{A}}_\ell + \frac{1-\eta}{B} \mathbf{Y}_\ell^\top \mathbf{Y}_\ell$$
$$\hat{\mathbf{S}}_\ell \leftarrow \eta \hat{\mathbf{S}}_\ell + \frac{1-\eta}{B} \mathcal{D}\mathbf{Z}_\ell^\top \mathcal{D}\mathbf{Z}_\ell,$$

  where $\mathbf{Y}$ and $\mathbf{Z}$ denote the matrices of activations and pre-activations for a batch.

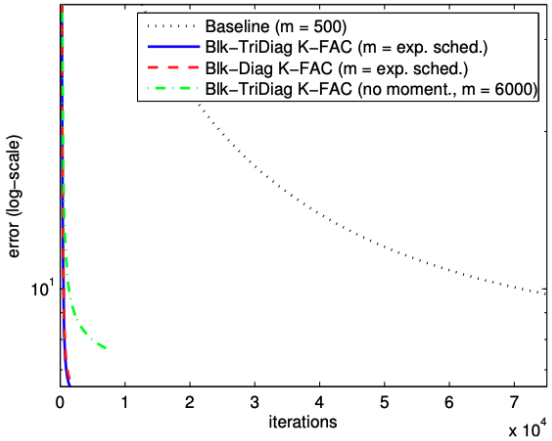
K-FAC: Odds and Ends

Some things we left out:

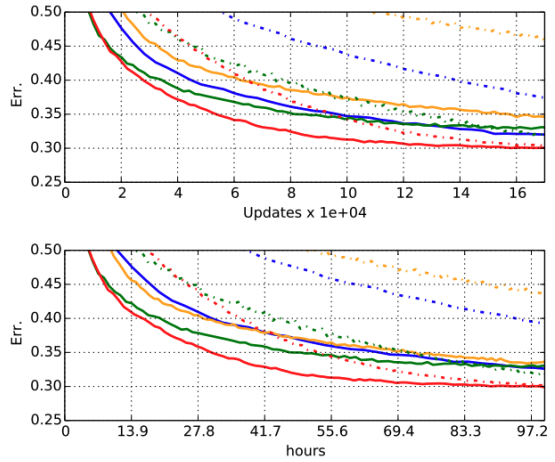
- Adding momentum and iterate averaging (straightforward)
- Using exact MVPs on the current batch to choose step sizes automatically
- Automatically adapting damping hyperparameters
- More accurate approximations than layerwise independence
- Extensions to other architectures (conv nets, RNNs, etc.)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

K-FAC: Results

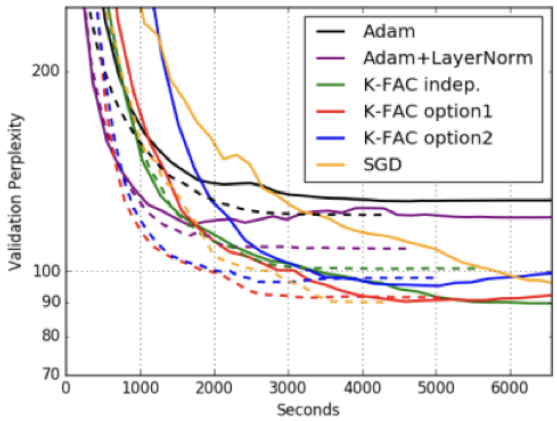
Logistic autoencoder



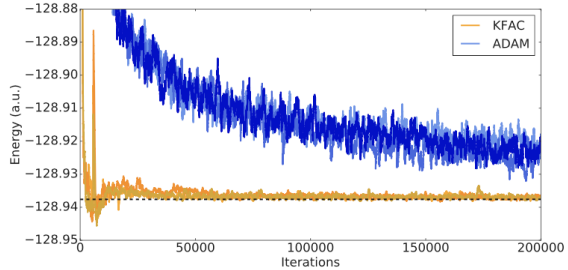
ImageNet classifier CNN



RNN language model



FermiNET (Schrödinger Equation)



- Results are sometimes amazing, sometimes meh.
- Why? Stay tuned for Lecture 7.