Course progress

- Previously
  - PCA
  - OLS, ridge regression, conjugate gradient
  - Convex optimization, linear programming, Lagrange multipliers, duality and minimax games
  - Sparse regression (LASSO); NMF, Sparse PCA,
  - Dual ascent, dual decomposition, augmented Lagrangians, ADMM
  - Random sampling and randomized QR and SVD factorizations
  - Compressed sensing and matrix completion
  - DFT and FFT, shift invariant and circulant matrices/2D Fourier transform/filters
  - Graphs and their matrix representation, clustering
  - *Stochastic gradient descent, classification models and neural networks, CNNs*
- Today: Backpropagation and hyperparameters (Sec. VII.3 and VII.4)

NN models

- Training data $(v_1, y_1)...(v_n, y_n)$ for $v_i \in \mathbb{R}^m$ and labels y_i.
- Minimize

$$L(x) = \frac{1}{n}\sum_{i=1}^{n} \ell_i(x)$$

- The loss function is evaluated, e.g., cross-entropy loss for k-category classification, is evaluated at i-th data point

$$\ell_i(x) = \sum_{j=1}^{k} y_{j,i} \log F_j(x, v_i)$$

where $F(x, v)$ is given by the neural network,

$$\begin{aligned} v_0 &= v, \\ v_{k+1} &= R(A_{k+1}v_k + b_{k+1}) \\ F(x, v) &= v_K \end{aligned}$$

and $x = (A_{1:K}, b_{1:K})$ represents all of its parameters.

NN models - weights $A_{1:K}$

- *Fully connected NN*
- Learning images
  - *CNN*
- Learning sequences
  - RNN, LSTM, GRU, Transformers
- Learning graphs
  - GNN

Gradient-based learning

- To learn x, we can follow a gradient-based descent algorithm

$$x_{k+1} = x_k - s_k \nabla L_{B_k}(x_k)$$

  where the gradient over batch B_k is

$$\nabla_{B_k} L(x_k) = \frac{1}{|B_k|} \sum_{i \in B_k} \nabla \ell_i(x_k)$$

- In the GD we take the gradient of the full batch, i.e., B_k is always all of the n samples
- Minibatch GD chooses a batch of size of $B < n$ uniformly at random
- In SGD, $|B| = 1$, and the algorithm chooses a single $i(k)$ at step k uniformly at random
- Alternatively, in practice, the data is randomly ordered and the algorithm goes through it sequentially batch by batch

Gradient computation

- By the univariate chain rule

$$\frac{\partial}{\partial x_j}\ell = \frac{\partial}{\partial F}\ell(F)\frac{\partial F}{\partial x_j}$$

- Therefore

$$\nabla \ell(x) = \frac{\partial}{\partial F}\ell(F(x))\nabla F(x)$$

  where $\ell(x) := \ell_i(x)$ also depends on i-th true label y_i and $F(x) := F(x, v_i)$ also depends on the i-th feature vector v_i
- But since we're not differentiating with respect to y_i or v_i, we just treat them as parameters.

Chain rule

- In our earlier fully connected NN example

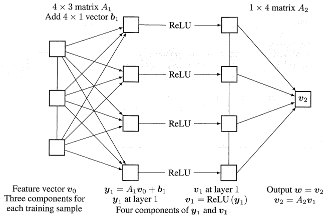
$$v_2 = F(A_k, b_k, v_0) = A_2 v_1 + b_2 = A_2(R(A_1 v_0 + b_1)) + b_2$$

where R is a set of 4 Relu activation functions
- The parameters or weights are

$$x = (A_1, b_1, A_2, b_2)$$

- Thus, we need a multivariate version of the chain rule. In 1D:

$$\frac{d}{dx}F_3(F_2(F_1(x))) = \left(\frac{dF_3}{dx}(F_2(F_1(x)))\right)\left(\frac{dF_2}{dx}(F_1(x))\right)\left(\frac{dF_1}{dx}(x)\right)$$



4×3 matrix A_1
Add 4×1 vector $\boldsymbol{b}_1$
1×4 matrix A_2
ReLU
ReLU
ReLU
ReLU
$\boldsymbol{v}_2$
Feature vector $\boldsymbol{v}_0$
Three components for each training sample
$\boldsymbol{y}_1 = A_1\boldsymbol{v}_0 + \boldsymbol{b}_1$
$\boldsymbol{y}_1$ at layer 1
$\boldsymbol{v}_1$ at layer 1
$\boldsymbol{v}_1 = \text{ReLU}(\boldsymbol{y}_1)$
Four components of $\boldsymbol{y}_1$ and $\boldsymbol{v}_1$
Output $\boldsymbol{w} = \boldsymbol{v}_2$
$\boldsymbol{v}_2 = A_2\boldsymbol{v}_1$

Figure: Fig VII.2 from [1]

Chain rule

- The chain rules in the multivariate setting is $D(f \circ g) = Df \circ Dg$, so

$$\frac{\partial}{\partial A}[R(Av+b)] = \frac{\partial R}{\partial u} \circ \frac{\partial}{\partial A}(Av+b)$$

- Here $\frac{\partial R}{\partial u}$ is just the Jacobian matrix

$$\begin{bmatrix} \frac{\partial R_1}{\partial u_1} & \cdots & \frac{\partial R_1}{\partial u_n} \\ \vdots & \ldots & \vdots \\ \frac{\partial R_m}{\partial u_1} & \ldots & \frac{\partial R_m}{\partial u_n} \end{bmatrix}$$

- For $R(u) = Relu(u) = \max(0, u)$, it is just the diagonal matrix

$$\frac{\partial R_i}{\partial u_i} = \begin{cases} 0 & \text{if } (Av+b)_i < 0 \\ 1 & \text{if } (Av+b)_i > 0 \end{cases}$$

- What is the derivative of $u = Av$ w/r/t the matrix A?

Derivative of matrix vector product

- What is the derivative of $u = Av$ w/r/t the matrix A?
- This is an order 3 tensor: since

$$u_t = \sum_\ell A_{t\ell} v_\ell$$

we have

$$\frac{\partial u_t}{\partial A_{jk}} = \frac{\partial}{\partial A_{jk}} \left(\sum_\ell A_{t\ell} v_\ell \right) = v_k \delta_{tj}$$

- Therefore,

$$\frac{\partial R_i}{\partial A_{jk}} = \sum_t \frac{\partial R_i}{\partial u_t} \frac{\partial u_t}{\partial A_{jk}} = \sum_t \frac{\partial R_i}{\partial u_t} v_k \delta_{tj} = \frac{\partial R_i}{\partial u_j} v_k$$

- In this fashion can automatically compute all the derivatives going backwards (called autodiff or backprop)

Chain rule - order of multiplication

- The associativity of matrix multiplication gives two choices to compute ABC: either $(AB)C$ or $A(BC)$
- It is easy to see that for square matrices

  $$M_1 M_2 w \text{ needs } N^3 + N^2 \text{ multiplications}$$

  while

  $$M_1(M_2 w) \text{ needs } N^2 + N^2 \text{ multiplications}$$

  (Item N^3 is a simplification as are in fact subcubic algorithms)
- So forward propagation

  $$(M_1 M_2) M_3) ... M_L) w \text{ needs } (L-1)N^3 + N^2 \text{ multiplications}$$

  while backwards

  $$M_1(M_2(... M_L w) \text{ needs } LN^2 \text{ multiplications}$$

  (Item N^3 is a simplification as are in fact subcubic algorithms)

Chain rule - order of multiplication

- The associativity of matrix multiplication gives two choices to compute ABC: either $(AB)C$ or $A(BC)$
- Let's say $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ The first way $(AB)C$

$$\begin{aligned} AB = (m \times n)(n \times q) &\text{ has } mnp \text{ multiplications} \\ (AB)C = (m \times p)(p \times q) &\text{ has } mpq \text{ multiplications} \\ &\text{Total } mp(n+q) \end{aligned}$$

- The second way $A(BC)$

$$\begin{aligned} &BC = (n \times p)(p \times q) \text{ has } npq \text{ multiplications} \\ &A(BC) = (m \times n)(n \times q) \text{ has } mnq \text{ multiplications} \\ &\text{Total } nq(m+p) \end{aligned}$$

Chain rule - order of multiplication

- If C is a column vector, $q = 1$, so $A(BC)$ has $n(m+p)$ steps vs $mnp + mp$ number of steps in $(AB)C$
- More generally we compare

$$\frac{mp(n+q)}{mnpq} = \frac{1}{q} + \frac{1}{n} \quad \text{vs} \quad \frac{nq(m+p)}{mnpq} = \frac{1}{m} + \frac{1}{p}$$

Chain rule

- Let's go back our feedforward fully connected NN example

$$w = v_2 = A_2 v_1 + b_2 = A_2(w(A_1 v_0 + b_1)) + b_2$$

- $\frac{\partial w}{\partial A_2}$ are given by

$$\frac{\partial w_i}{\partial A_{2jk}} = v_{1,k}\delta_{ij}$$

- and $\frac{\partial w}{\partial b_2}$ are

$$\frac{\partial w_i}{\partial b_{2j}} = \delta_{ij}$$

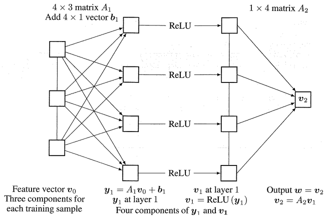

4×3 matrix A_1
Add 4×1 vector $\boldsymbol{b}_1$
1×4 matrix A_2
ReLU
ReLU
ReLU
ReLU
$\boldsymbol{v}_2$
Feature vector $\boldsymbol{v}_0$
Three components for each training sample
$\boldsymbol{y}_1 = A_1\boldsymbol{v}_0 + \boldsymbol{b}_1$
$\boldsymbol{y}_1$ at layer 1
$\boldsymbol{v}_1$ at layer 1
$\boldsymbol{v}_1 = \text{ReLU}(\boldsymbol{y}_1)$
Four components of $\boldsymbol{y}_1$ and $\boldsymbol{v}_1$
Output $\boldsymbol{w} = \boldsymbol{v}_2$
$\boldsymbol{v}_2 = A_2\boldsymbol{v}_1$

Figure: Fig VII.2 from [1]

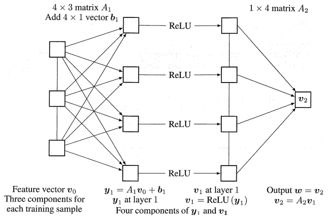
Chain rule

- From the previous slide

$$w = v_2 = A_2 v_1 + b_2 = A_2(R(A_1 v_0 + b_1)) + b_2$$

- By the chain rule

$$\frac{\partial w}{\partial A_1} = \frac{\partial A_2(R(A_1 v_0 + b_1))}{\partial A_1} = A_2(R'(A_1 v_0 + b_1)) \frac{\partial (A_1 v_0 + b_1)}{\partial A_1}$$



4×3 matrix A_1
Add 4×1 vector $\boldsymbol{b}_1$
1×4 matrix A_2
ReLU
ReLU
ReLU
ReLU
$\boldsymbol{v}_2$
Feature vector $\boldsymbol{v}_0$
Three components for each training sample
$\boldsymbol{y}_1 = A_1 \boldsymbol{v}_0 + \boldsymbol{b}_1$
$\boldsymbol{y}_1$ at layer 1
$\boldsymbol{v}_1$ at layer 1
$\boldsymbol{v}_1 = \text{ReLU}(\boldsymbol{y}_1)$
Four components of $\boldsymbol{y}_1$ and $\boldsymbol{v}_1$
Output $\boldsymbol{w} = \boldsymbol{v}_2$
$\boldsymbol{v}_2 = A_2 \boldsymbol{v}_1$

Figure: Fig VII.2 from [1]

Adjoint methods

- The above notion that backpropagation optimizes the computation of derivatives is exploited in other large scale optimization problems.
- Let's say we want to solve

$$Ev = b(p)$$

- The solution vector

$$v(p) = E^{-1}b(p)$$

also depends on p and let's say the Jacobian $\frac{\partial v}{\partial p}$ is $N \times M$

Adjoint methods

- Taking the derivatives of both sides

$$Ev(p) = b(p)$$

- gives

$$E\frac{\partial v}{\partial p_j} = \frac{\partial b}{\partial p_j} \text{ for } j = 1, ..., M$$

so

$$\frac{\partial v}{\partial p} = E^{-1}\frac{\partial b}{\partial p}$$

it seems that we have M linear systems of size N which will be expensive to solve iteratively as we minimize some loss function

$$F(v(p))$$

w/r/t p.

Adjoint methods

- If $F(v) = c^T v$ is linear, then

$$\frac{\partial F}{\partial p} = \frac{\partial F}{\partial v}\frac{\partial v}{\partial p} = c^T E^{-1}\frac{\partial b}{\partial p}$$

- Here multiply a row vector is multiplied by an $N \times N$ matrix E^{-1} and then by an $N \times M$ matrix $\frac{\partial b}{\partial p}$
- So you want to compute $(c^T E^{-1})\frac{\partial b}{\partial p}$.
- The first step is equivalent to solving the adjoint equation

$$c = E^T\lambda \;\Rightarrow\; c^T = \lambda^T E \;\Rightarrow\; \lambda^T = c^T E^{-1}$$

- $$\frac{\partial F}{\partial p} = \lambda^T\frac{\partial b}{\partial p}$$

  which entails multiplying a row vector is multiplied by an $N \times M$

Gradient-based learning

- To learn x, i.e. find x such that $L(x)$ is a (possibly local) minimum, follow a gradient-based descent algorithm

$$x_{k+1} = x_k - s_k \nabla L_{B_k}(x_k)$$

where the gradient over batch B_k is

$$\nabla_{B_k} L(x_k) = \frac{1}{|B_k|} \sum_{i \in B_k} \nabla \ell_i(x_k)$$

- In the GD we take the gradient of the full batch, i.e., B_k is always all of the n samples
- Minibatch GD chooses a batch of size of $B < n$ uniformly at random
- In SGD, $|B| = 1$, and the algorithm chooses a single $i(k)$ at step k uniformly at random

Learning rate

- Several ways to determine s_k depending on the algorithm
- For GD, previous guarantee applied for a fixed step size $s \leq 1/M$ where M is the Lipschitz constant of the gradient

$$\|\nabla f(x) - \nabla f(y)\| \leq M\|x - y\|$$

  uniformly over the domain (or a closed subset of the domain that includes the initialization and the minimum)
- Equivalently, if f is C^2 the eigenvalues λ_i of the Hessian of f are $|\lambda_i| \leq M$ for all i uniformly in x
- Often M is not known, but we can extend the convergence guarantees to *exact line search*

$$s_k = \arg\min_{s \geq 0} f(x_k - s\nabla f(x_k))$$

- and *backtracking line search*, which entails iteratively reducing s_k until

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2}s_k\|\nabla f(x_k)\|_2^2$$

Gradient decent convergence - fixed step size

- The previous argument didn't assume that f is convex, so the GD was converging to a local minimum.
- Finding a global minimum requires random x_k or grid search, which requires $t = O(1/\epsilon^d)$ iterations to achieve $\|\nabla f(x_k)\|_2^2 \leq \epsilon$ for functions with Lipschitz continuous gradients and $x \in \mathbb{R}^d$
- In practice gradient-based methods work well for non-convex functions used in NN even though there are not theoretical convergence guarantees
- If f is convex, then $\nabla f(x^*) = 0$ at a global minimizer x^*
- Therefore the above argument guarantees convergence to the global optimum at the above rate

Gradient decent convergence - fixed step size

- If f is *strongly* convex, i.e, the eigenvalues λ_i of the Hessian H are also $0 < m \le \lambda_i$ uniformly in x, the convergence $O((1 - \frac{m}{M})^k)$ for $0 < c < 1$.
- This means that a bound of

$$f(x_k) - f(x^*) \le \epsilon$$

  can be achieved using only $O(\log(1/\epsilon))$ iterations. See previous lecture on convex optimization
- This rate is called "linear convergence" for historic reasons (the error lies below a line on a log-linear plot of the error vs iteration number)
- Typical loss functions in NN are not strongly convex, e.g., Relu and softmax are convex but not strongly convex
- Adding an ℓ^2 regularization term will make them such and can improve convergence

Nesterov accelerated descent

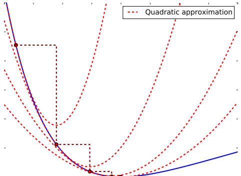
- If f is convex (but not necessarily strongly convex) is the $t = O(1/\epsilon)$ "sublinear convergence" optimal?
- So called Nesterov accelerated descent

  $$\begin{aligned} x_{k+1} &= y_k - s\nabla f(y_k) \\ y_{k+1} &= x_{k+1} + \beta_k(x_{k+1} - x_k) \end{aligned}$$

  achieves error of $O(1/t^2)$ after t iterations.
- Can use $s = 1/M$ and $\beta_k = (k-1)/(k+2)$
- So only needs $t = O(1/\sqrt{\epsilon})$ to get within ϵ of the solution
- It not straightforward to understand why this method works better
- One observation is that this is not a descent method, i.e., the steps may overshoot the minimum and oscillate around it, rather than converging from one direction.
- Used in practice to optimize convex and nonconvex function in ML

Second order methods: Newton's method

- A second order approximation of a convex C^2 function is
$$g(y) = f(x) + \nabla f(y - x) + \frac{1}{2}(y - x)^T \nabla^2 f(x)(y - x)$$
- If the Hessian is positive definite,
$$\arg\min_y g(y) = x - (\nabla^2 f(x))^{-1} \nabla f(x)$$
- This idea leads to Newton's method
$$x_{k+1} = x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$$
which has quadratic convergence under certain assumptions
- Rarely used in DL since Hessian computation is more computationally expensive, and the convergence of NN is only guaranteed when you're close to an optimal point



SGD step size

- In previous lecture, we showed sublinear convergence of SGD with a fixed step size

$$\min_{1\leq k\leq t} \mathbb{E}\|\nabla f(x_k)\|^2 \leq \frac{C}{\sqrt{t}}$$

- So the norm is below ϵ if

$$\frac{C}{\sqrt{t}} \leq \epsilon$$

- This is guaranteed for

$$\frac{C^2}{\epsilon^2} \leq t$$

- So SGD requires $t = O(1/\epsilon^2)$ iterations to achieve $\mathbb{E}\|\nabla f(x_k)\|_2^2 \leq \epsilon$.
- vs GD that requires $t = O(1/\epsilon)$

SGD step size

- In practice reduce the step size as the optimization continues ($s_k \sim \frac{c}{k}$ or $\frac{c}{\sqrt{k}}$)
  - Convergence with probability 1 guarantees - Robbins Siegmund theorem
  - When the learning rates decrease too quickly, the expectation of the estimate takes too long to approach the optimum
  - When the learning rates decrease too slowly, the variance of the SGD estimate reduces to slowly
- When the Hessian of the objective is PD, can speed up the convergence of the expectation, but this does not reduce the variance
- Even though the convergence rate is slower, the per iteration time of SGD is faster
- Use SGD when the training time is the bottleneck. See Leon Bottou, *SGD Tricks*

Cross-validation

- K-fold cross validation
- Randomly spilt the data into K sets (epochs).
- Use one set as a training set and $K - 1$ as test sets.
- Repeat the procedure for different learning rates
- Find the best training set and the best learning rate that minimize the test error

Batch normalization

- We want the data in every minibatch to have uniform mean and variance.
- Let $v_1, ...v_B$ be the size of the minibatch
- $V_i = (v_i - \mu)/\sqrt{\sigma^2 + \epsilon}$ for small $\epsilon > 0$, and sample mean μ and variance σ^2.
- The input is
$$y_i = \gamma V_i + \beta$$
where γ and β are trainable parameters.

Dropout and regularization

- To avoid overfitting, for each data point v_i randomly dropout connections in NN according to Bernoulli distribution
- Leads to fewer parameters in each case but the full architecture is still available on average.
- L1 and L2 regularization with coefficients determined by cross-validation can be also used

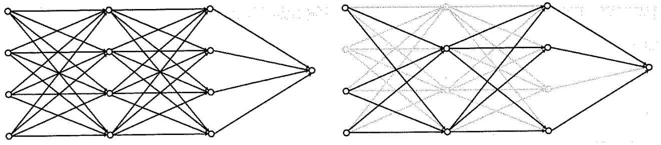


Figure: Fig VII.6 from [1])

Next steps

- RNNs, LSTMs, GRUs, Transformers
- GNNs
- Kernel methods
- Bandit problems and reinforcement learning