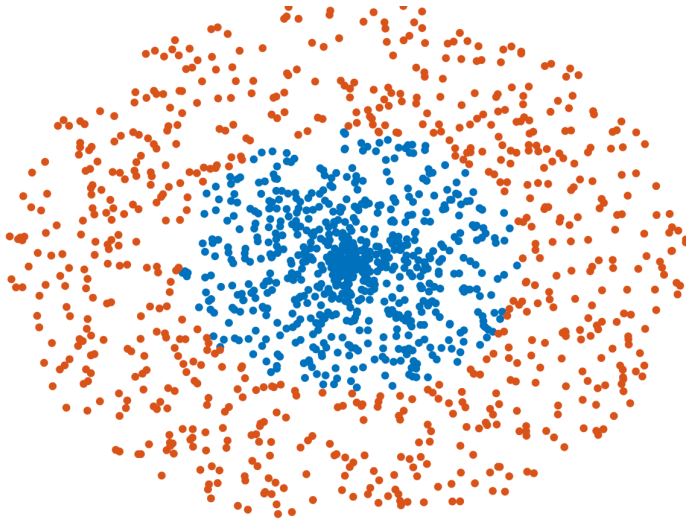
Single-Layer Neural Networks

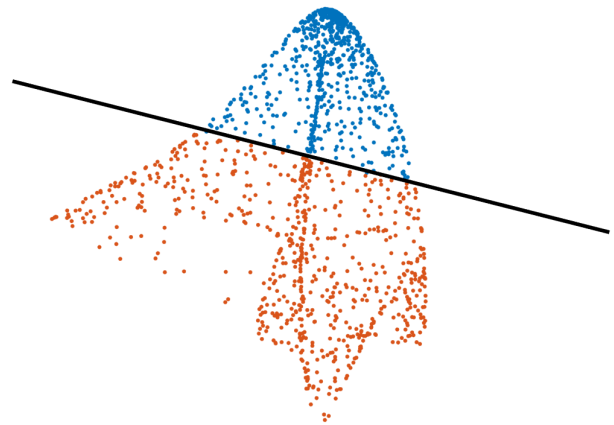
Numerical Methods for Deep Learning

Motivation: Nonlinear Models

In general, impossible to find a linear separator between classes



input features



transformed features

Goal/Trick

Embed the points in higher dimension and/or move the points to make them linearly separable

Learning Objective: Single-Layer Neural Networks

In this module, we derive our first nonlinear model, i.e., a neural network with a single layer.

Learning tasks:

- classification $\rightsquigarrow$ multinomial logistic regression
- regression $\rightsquigarrow$ nonlinear least-squares

Numerical methods:

- Sample Average Approximation: Newton-CG, VarPro, ...
- Stochastic Optimization: SGD, ADAM, ...

Example: Linear Regression

Assume $\mathbf{C} \in \mathbb{R}^{n_c \times n}$, $\mathbf{Y} \in \mathbb{R}^{n_f \times n}$ and $n \gg n_f$. Goal: Find $\mathbf{W} \in \mathbb{R}^{n_c \times n_f}$ such that

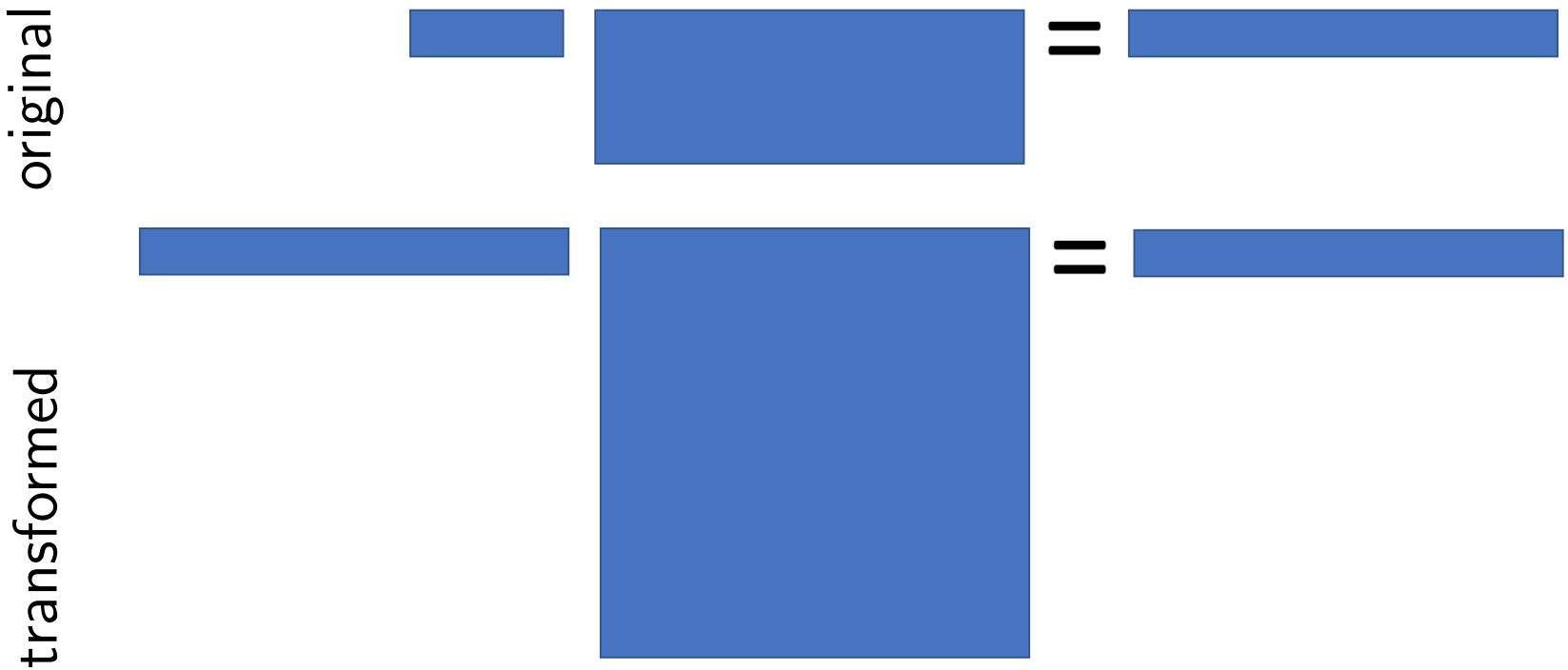
$$\mathbf{C} = \mathbf{W}\mathbf{Y}$$

Since $\mathrm{rank}(\mathbf{Y}) < n$, there will generally be no solution.

Two options:

1. Regression: Solve $\min_{\mathbf{W}} \|\mathbf{W}\mathbf{Y} - \mathbf{C}\|_F^2 \rightsquigarrow$ always has solutions, but residual might be large
2. Nonlinear Model: Replace $\mathbf{Y}$ by $\sigma(\mathbf{K}\mathbf{Y})$ in regression, where σ is element-wise function (aka activation) and $\mathbf{K} \in \mathbb{R}^{m \times n_f}$ where $m \gg n_f$

Illustrating Nonlinear Models



Remarks

- instead of $\mathbf{WY} = \mathbf{C}$ solve $\hat{\mathbf{W}}\sigma(\mathbf{KY}) = \mathbf{C}$
- solve bigger problem $\rightsquigarrow$ memory, computation, ...
- what happens to $\mathrm{rank}(\sigma(\mathbf{KY}))$ when $\sigma(x) = x$?

Conjecture: Universal Approximation Properties

Given the data $\mathbf{Y} \in \mathbb{R}^{n_f \times n}$ and $\mathbf{C} \in \mathbb{R}^{n_c \times n}$ with $n \gg n_f$, there is a nonlinear function $\sigma : \mathbb{R} \to \mathbb{R}$, a matrix $\mathbf{K} \in \mathbb{R}^{m \times n_f}$, and a bias $\mathbf{b} \in \mathbb{R}^m$ such that

$$\operatorname{rank}(\sigma(\mathbf{KY} + \mathbf{be}_n^\top)) = n.$$

Therefore, possible to find $\mathbf{W} \in \mathbb{R}^{n_c \times m}$

$$\mathbf{W}\sigma(\mathbf{KY} + \mathbf{be}_n^\top) = \mathbf{C}.$$

This is only a conjecture. For solid approximation theory see [4, 7].

Choosing Nonlinear Model

$$\mathbf{W}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top) = \mathbf{C}$$

- how to choose σ?
  - early days: motivated by neurons
  - popular choice: $\sigma(x) = \tanh(x)$ (smooth, bounded, ...)
  - nowadays: $\sigma(x) = \max(x, 0)$ (aka ReLU, rectified linear unit, non-differentiable, not bounded, simple)
- how to choose $\mathbf{K}$ and $\mathbf{b}$?
  - pick randomly $\rightsquigarrow$ branded as *extreme learning machines* [8]
  - train (optimize) $\rightsquigarrow$ done for most neural network
  - *deep learning* when neural network has many layers

Extreme Learning Machines [8]

Select activation function, choose $\mathbf{K}$ and $\mathbf{b}$ randomly, and solve the linear least-squares/classification problem.

Advantages:

- universal approximation theorem: can interpolate any function
- very(!) easy to program, convex optimization
- can serve as a benchmark to more sophisticated methods

Some concerns:

- may require very large $\mathbf{K}$ (scale with n, number of examples)
- may not generalize well
- large-scale optimization problem with no obvious structure

Today: Learning the Weights

Why? Using random weights, $\mathbf{K}$ might need to be very large to fit training data (scales with n).
Also, solution may not generalize well to test data.

Idea: Learn $\mathbf{K}$ and $\mathbf{b}$ from the data (in addition to $\mathbf{W}$)

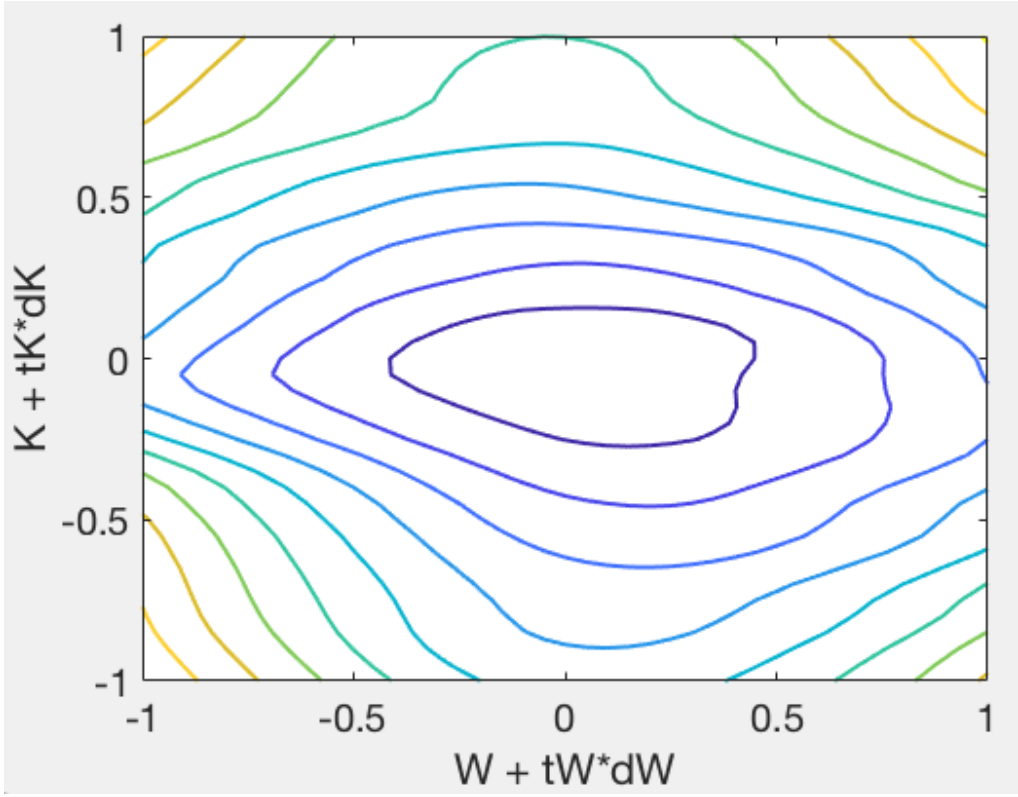
$$\min_{\mathbf{K},\mathbf{W},\mathbf{b}} E(\mathbf{W}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top), \mathbf{C}_{\text{obs}}) + \lambda R(\mathbf{W}, \mathbf{K}, \mathbf{b})$$

About this optimization problem:

- unknowns $\mathbf{W} \in \mathbb{R}^{n_c \times m}$, $\mathbf{K} \in \mathbb{R}^{m \times n_f}$, $\mathbf{b} \in \mathbb{R}^m$
- new hyper-parameter m (aka width, number of neurons)
- non-convex problem $\rightsquigarrow$ local minima, careful initialization
- need to compute derivatives w.r.t. $\mathbf{K}, \mathbf{b}$

Non-Convexity

The optimization problem is non-convex. Simple illustration of cross-entropy along two random directions $d\mathbf{K}$ and $d\mathbf{W}$



Expect worse when number of layers grows!

Recap: Differentiating Linear Algebra Expressions

Easy ones:

$$F_1(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top \mathbf{y} \qquad \mathbf{J}_\mathbf{x} F_1(\mathbf{x}, \mathbf{y}) = \mathbf{y}^\top$$
$$F_2(\mathbf{A}, \mathbf{x}) = \mathbf{A}\mathbf{x} \qquad \mathbf{J}_\mathbf{x} F_2(\mathbf{x}, \mathbf{y}) = \mathbf{A}$$

For $\mathbf{x} = \mathrm{vec}(\mathbf{X})$ what is

$$F_3(\mathbf{A}, \mathbf{X}) = \mathbf{A}\mathbf{X} \qquad \mathbf{J}_\mathbf{x} F_3 = ???$$

Recall that

$$\mathrm{vec}(\mathbf{A}\mathbf{X}) = \mathrm{vec}(\mathbf{A}\mathbf{X}\mathbf{I}) = (\mathbf{I} \otimes \mathbf{A})\mathrm{vec}(\mathbf{X})$$

Therefore:

$$\mathbf{J}_\mathbf{x} F_3(\mathbf{A}, \mathbf{X}) = \mathbf{I} \otimes \mathbf{A}$$

Efficient mat-vec: $\mathbf{J}_\mathbf{X} F_3(\mathbf{A}, \mathbf{X})\mathbf{v} = \mathbf{A}\ \mathrm{mat}(\mathbf{v})$

Training Single Layer Neural Network

Assume no regularization (easy to add) and re-write optimization problem as

$$\min_{\mathbf{W},\mathbf{K},\mathbf{b}} E(\mathbf{C}_{\text{obs}}, \mathbf{Z}, \mathbf{W}) \quad \text{with} \quad \mathbf{Z} = \sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top)$$

Agenda:

1. compute derivative of $\mathbf{z} = \text{vec}(\mathbf{Z})$ w.r.t. $\text{vec}(\mathbf{K}), \mathbf{b}$
2. use chain rule to get

$$\mathbf{J}_{\text{vec}(\mathbf{K})} E = \mathbf{J}_{\text{vec}(\mathbf{Z})} E(\mathbf{C}_{\text{obs}}, \mathbf{Z}, \mathbf{W})\ \mathbf{J}_{\text{vec}(\mathbf{K})} \mathbf{Z}$$
$$\mathbf{J}_{\mathbf{b}} E = \mathbf{J}_{\text{vec}(\mathbf{Z})} E(\mathbf{C}_{\text{obs}}, \mathbf{Z}, \mathbf{W})\ \mathbf{J}_{\mathbf{b}} \mathbf{Z}$$

3. efficient code for mat-vecs with $\mathbf{J}$ and $\mathbf{J}^\top$

Derivatives of a Single Layer Network

$$\mathbf{Z} = \sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top)$$

Recall that σ is applied element-wise. Therefore

$$\mathbf{J}_{\mathrm{vec}(\mathbf{K})}\mathbf{Z} = \mathrm{diag}(\sigma'(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top))(\mathbf{Y}^\top \otimes \mathbf{I})$$

Efficient way to get matrix vector products

$$\begin{aligned} \mathbf{J}_\mathbf{K}\mathbf{Z}\mathbf{v} &= \mathrm{mat}\left(\mathrm{diag}(\sigma'(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top)\ (\mathbf{Y}^\top \otimes \mathbf{I})\mathbf{v}\right) \\ &= \sigma'(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top) \odot (\mathrm{mat}(\mathbf{v})\mathbf{Y}) \end{aligned}$$

And for transpose

$$\begin{aligned} (\mathbf{J}_\mathbf{K}\mathbf{Z})^\top\mathbf{u} &= \mathrm{mat}\left((\mathbf{Y} \otimes \mathbf{I})\ \mathrm{diag}(\sigma'(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top))\mathbf{u}\right) \\ &= (\sigma'(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top) \odot \mathrm{mat}(\mathbf{u}))\ \mathbf{Y}^\top \end{aligned}$$

Coding Problem: Derivatives of Single Layer

Derivations:

1. compute $\mathbf{J_b Z v}$ and $(\mathbf{J_b Z})^\top \mathbf{u}$
2. (optional) compute $\mathbf{J}_{\mathrm{vec}(\mathbf{Y})}\mathbf{Z v}$ and $(\mathbf{J}_{\mathrm{vec}(\mathbf{Y})}\mathbf{Z})^\top \mathbf{u}$

Coding:

```
function[Z,JKt,Jbt,JYt,JK,Jb,JY] = singleLayer(K,b,Y)
% Returns Z = sigma(K*Y+b) and
%                              functions for J'*U and J*V
```

Testing:

1. Derivative check for Jacobian mat-vec
2. Adjoint tests for transpose, let $\mathbf{v}, \mathbf{u}$ be arbitray vectors

$$\mathbf{u}^\top \mathbf{J v} \approx \mathbf{v}^\top \mathbf{J}^\top \mathbf{u}$$

Putting Things Together

Implement loss function of single-layer NN

$$E(\mathbf{K}, \mathbf{b}, \mathbf{W}) \stackrel{def}{=} E(\mathbf{C}, \mathbf{Z}, \mathbf{W}), \quad \mathbf{Z} = \sigma(\mathbf{KY} + \mathbf{be}_n^\top)$$

```
function [Ec,dE] = singleLayerNNObjFun(x,Y,C,m)
% where x = [K(:); b; W(:)]
% evaluates single layer and computes cross entropy
%         and gradient (extend for approx. Hessian)
```

Use

1. $\nabla_{\mathbf{Z}} E = \mathbf{W}^\top \nabla_{\mathbf{S}}\, E(\mathbf{S}), \quad \mathbf{S} = \mathbf{WZ}$
2. $\nabla_{\mathbf{K}} E = \mathbf{J}_{\mathbf{K}}^\top \nabla_{\mathbf{Z}} E$
3. $\nabla_{\mathbf{b}} E = \mathbf{J}_{\mathbf{b}}^\top \nabla_{\mathbf{Z}} E$
4. $\nabla_{\mathbf{W}} E = \nabla_{\mathbf{S}}\, E(\mathbf{S})\mathbf{Y}$

Sample Average Approximation (SAA)

Note that the objective function in our learning problem is actually stochastic

$$\frac{1}{n} E(\mathbf{W}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top), \mathbf{C}_{\text{obs}}) = \mathbb{E}_{(\mathbf{y},\mathbf{c})}\left[E\left(\mathbf{W}\sigma(\mathbf{K}\mathbf{y} + \mathbf{b}), \mathbf{c}\right)\right]$$

In general, n will be too large to compute left hand side $\rightsquigarrow$ consider stochastic problem.

SAA idea: Approximate expected value with relatively large sample $S \subset \{1, \ldots, n\}$. Use deterministic optimization method

$$\min_{\mathbf{K},\mathbf{b},\mathbf{W}} \frac{1}{|S|} \sum_{s \in S} E\left(\mathbf{W}\sigma(\mathbf{K}\mathbf{y} + \mathbf{b}), \mathbf{c}\right).$$

Pro: use your favorite solver, linesearch, stopping...
Con: large batches needed

Note: Sample stays fixed during iteration, but occasional resampling recommended.

Simple Option: BFGS, NLCG, ...

Since we have computed the gradient of our objective function, we can experiment with a wide range of methods already.

Some candidates from `scipy.optimize.minimize` are:

- `CG` - nonlinear conjugate gradient
- `BFGS`
- `Newton-CG` - attention: Hessian not spsd
- `trust-ncg`

Note that for the latter two, Hessian mat-vecs will be approximated numerically (not very stable).

Better Option: Gauss-Newton Method

Goal: Use curvature information for fast convergence

$$\nabla_{\mathbf{K}} E(\mathbf{K}, \mathbf{b}, \mathbf{W}) = (\mathbf{J}_{\mathbf{K}}\mathbf{Z})^\top \nabla_{\mathbf{Z}} E(\mathbf{W}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top), \mathbf{C}),$$

where $\mathbf{J}_{\mathbf{K}}\mathbf{Z} = \nabla_{\mathbf{K}}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top)^\top$.This means that Hessian is

$$\begin{aligned}\nabla^2_{\mathbf{K}} E(\mathbf{K}) = (\mathbf{J}_{\mathbf{K}}\mathbf{Z})^\top \nabla^2_{\mathbf{Z}} E(\mathbf{C}, \mathbf{Z}, \mathbf{W}) \mathbf{J}_{\mathbf{K}}\mathbf{Z} \\ + \sum_{i=1}^{n}\sum_{j=1}^{m} \nabla^2_{\mathbf{K}}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top)_{ij} \nabla_{\mathbf{Z}} E(\mathbf{C}, \mathbf{Z}, \mathbf{W})_{ij}\end{aligned}$$

First term is spsd and we can compute it.
We neglect second term since

- can be indefinite and difficult to compute
- small if transformation is roughly linear or close to solution (easy to see for least-squares)

same for $\mathbf{b}$ and use full Hessian for $\mathbf{W} \rightsquigarrow$ ignore coupling!

Even Better Option: Variable Projection [9]

Idea: Treat learning problem as coupled optimization problem with blocks $\boldsymbol{\theta}$ and $\mathbf{W}$.
Simple illustration for coupled least-squares problem [6, 5, 10]

$$\min_{\boldsymbol{\theta},\mathbf{w}} J(\boldsymbol{\theta},\mathbf{w}) = \frac{1}{2}\|\mathbf{A}(\boldsymbol{\theta})\mathbf{w}-\mathbf{c}\|^2 + \frac{\lambda}{2}\|\mathbf{L}\mathbf{w}\|^2 + \frac{\beta}{2}\|\mathbf{M}\boldsymbol{\theta}\|^2$$

Note that for given $\boldsymbol{\theta}$ the problem becomes a standard least-squares problem. Define:

$$\mathbf{w}(\boldsymbol{\theta}) = \left(\mathbf{A}(\boldsymbol{\theta})^\top\mathbf{A}(\boldsymbol{\theta}) + \lambda\mathbf{L}^\top\mathbf{L}\right)^{-1}\mathbf{A}(\boldsymbol{\theta})^\top\mathbf{c}$$

This gives optimization problem in $\boldsymbol{\theta}$ only (aka *reduced/projected problem*)

$$\min_{\boldsymbol{\theta}} \tilde{J}(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{A}(\boldsymbol{\theta})\mathbf{w}(\boldsymbol{\theta})-\mathbf{c}\|^2 + \frac{\lambda}{2}\|\mathbf{L}\mathbf{w}(\boldsymbol{\theta})\|^2 + \frac{\beta}{2}\|\mathbf{M}\boldsymbol{\theta}\|^2$$

Variable Projection (cont.)

$$\min_{\boldsymbol{\theta}} \tilde{J}(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{A}(\boldsymbol{\theta})\mathbf{w}(\boldsymbol{\theta}) - \mathbf{c}\|^2 + \frac{\lambda}{2}\|\mathbf{L}\mathbf{w}(\boldsymbol{\theta})\|^2 + \frac{\beta}{2}\|\mathbf{M}\boldsymbol{\theta}\|^2$$

Necessary optimality condition:

$$\nabla \tilde{J}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, \mathbf{w}) + \nabla_{\boldsymbol{\theta}} \mathbf{w}(\boldsymbol{\theta}) \nabla_{\mathbf{w}} J(\boldsymbol{\theta}, \mathbf{w}) \stackrel{!}{=} 0.$$

Less complicated than it seems since

$$\nabla_{\mathbf{w}} J(\boldsymbol{\theta}, \mathbf{w}(\boldsymbol{\theta})) = \mathbf{A}(\boldsymbol{\theta})^\top (\mathbf{A}(\boldsymbol{\theta})\mathbf{w}(\boldsymbol{\theta}) - \mathbf{c}) + \lambda \mathbf{L}^\top \mathbf{L}\mathbf{w}(\boldsymbol{\theta}) = 0$$

Discussion:

- ignore second term in gradient computation
- apply gradient descent/NLCG/BFGS to minimize $\tilde{J}$
- solve least-squares problem in each evaluation of $\tilde{J}$
- gradient is only correct if LS problem is solved exactly

Variable Projection for Single Layer

$$\min_{\mathbf{K},\mathbf{b},\mathbf{W}} E(\mathbf{W}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top), \mathbf{C}) + \lambda R(\boldsymbol{\theta}, \mathbf{W})$$

Assume that the regularizer is separable, i.e.,

$$R(\mathbf{K}, \mathbf{b}, \mathbf{W}) = R_1(\mathbf{K}, \mathbf{b}) + R_2(\mathbf{W})$$

and that R_2 is convex and smooth. Hence, the projection requires solving the regularized classification problem

$$\mathbf{W}(\mathbf{K}, \mathbf{b}) = \arg\min_{\mathbf{W}} E(\mathbf{W}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top), \mathbf{C}) + \lambda R_2(\mathbf{W})$$

practical considerations:

- solve for $\mathbf{W}(\mathbf{K}, \mathbf{b})$ using SVD, Newton (need accuracy)
- errors in $\mathbf{W}(\mathbf{K}, \mathbf{b}) \rightsquigarrow$ errors in $\nabla \tilde{J}(\mathbf{K})$, $\nabla \tilde{J}(\mathbf{b})$
- use gradient-based minimization to solve for $\mathbf{K}, \mathbf{b}$

Practical Considerations in SAA

Here is a simple but effective SAA-based training algorithm.

Pick $(\mathbf{K}_0, \mathbf{b}_0, \mathbf{W}_0)$ randomly and then do one or more steps of:

1. randomly select samples S (large enough)
2. take a few minimization steps
3. check and print training error on current batch and validation error
4. repeat

Possible problems:

- $|S|$ too small $\rightarrow$ training error small but no generalization
- $|S|$ too large $\rightarrow$ training too slow

Discussion: Sample Average Approximation

Idea: Approximate expected value with samples S

$$\frac{1}{|S|}\sum_{s\in S} E\left(\mathbf{W}\sigma(\mathbf{K}\mathbf{y}+\mathbf{b}),\mathbf{c}\right) \approx \mathbb{E}_{(\mathbf{y},\mathbf{c})}\left[E\left(\mathbf{W}\sigma(\mathbf{K}\mathbf{y}+\mathbf{b}),\mathbf{c}\right)\right]$$

Advantage: Can use deterministic gradient-based methods, e.g., steepest descent, nonlinear CG, BFGS, Gauss-Newton, VarPro, ...

Drawbacks:

- Evaluating gradient needs pass through the entire sample.
- Sample size must be large enough to avoid overfitting

Stochastic Approximation

Goal: minimize the expected loss

$$\mathbb{E}_{(\mathbf{y},\mathbf{c})}\left[E\left(\mathbf{W}\sigma(\mathbf{K}\mathbf{y}+\mathbf{b}),\mathbf{c}\right)\right]$$

Assume that each $\mathbf{y}_i$, $\mathbf{c}_i$ pair is drawn from some (unknown probability distribution). This is a stochastic optimization problem [3].

Examples: iterations $(\mathbf{K}_k, \mathbf{b}_k, \mathbf{W}_k) \to (\mathbf{K}^*, \mathbf{b}^*, \mathbf{W}^*)$ that (under certain conditions) decrease the expected value: Stochastic Gradient Descent, ADAM, ...

Pro: sample can be small (*mini batch*), often finds global minima for non-convex problems (not much theory though)

Con: how to monitor objective, linesearch, descent, ...

Stochastic Gradient Descent

Consider

$$\min_{\boldsymbol{\theta}} F(\boldsymbol{\theta}, \mathbf{Y}), \quad \text{with} \quad F(\boldsymbol{\theta}, \mathbf{Y}) = \frac{1}{n} \sum_{i=1}^{n} f_i(\boldsymbol{\theta}, \mathbf{y}_i).$$

Let $\mathcal{S}_k \subset \{1, 2, \ldots, n\}$. Define the batch objective function as

$$F_{\mathcal{S}_k}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} f_i(\boldsymbol{\theta}, \mathbf{Y}_i)$$

Then a straight forward extension is

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mu_k \mathbf{A}_k^{-1} \nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k)$$

Questions

- Would the method converge?
- Under what conditions on $\mu_k, \mathbf{A}_k, \mathcal{S}_k$?
- How fast?

References: original method [11], recent surveys [2, 1, 3]

Stochastic Gradient Descent

Let $\mathcal{S}_k \subset \{1, 2, \ldots, n\}$. Define the batch objective function as

$$F_{\mathcal{S}_k}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} f_i(\boldsymbol{\theta}, \mathbf{Y}_i)$$

Then a straight forward extension is

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mu_k \mathbf{A}_k^{-1} \nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k)$$

If f_i are convex, $\mathbf{A}_k = \mathbf{I}$, $|\mathcal{S}_k| = 1$ and $\mu_k \to 0$ slowly enough, that is

$$\sum_{k=1}^{\infty} \mu_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \mu_k^2 < \infty$$

then SGD converges to stationary point (Ex: $\mu_k = k^{-1}$).

How fast? Convergence is **sublinear**

A Glimpse into the theory

Consider the iteration and $\mathbf{A}_k = \mathbf{I}$

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mu_k \nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k)$$

Re-write this as

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \underbrace{\mu_k \nabla F(\boldsymbol{\theta}, \mathbf{Y})}_{\text{true gradient}} - \underbrace{\mu_k \left(\nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k) - \nabla F(\boldsymbol{\theta}, \mathbf{Y})\right)}_{\text{noise}}$$

Note that (unbiased estimator)

$$\mathbb{E}(\nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k)) = \nabla F(\boldsymbol{\theta}).$$

Finally note that

$$\mathrm{Var}\left(\mu_k \nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k)\right) = \mu_k^2 \mathrm{Var}\left(\nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k)\right)$$

Improvements of SGD: Momentum

Idea: Accelerate convergence by keeping gradient informations from previous batches.

$$\mathbf{S}_{k+1} = \gamma \mathbf{S}_k + \mu_k \nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{S}_{k+1}$$

μ_k - learning rate, γ - momentum

Hard to choose in practice, heuristic
γ - Start with 0.5 and increase slowly to 0.9
μ - problem dependent start small and decrease after a few epoch

Improvements of SGD: Nesterov

Idea: Predict next iterate using momentum, correct next step using gradient there.

$$\boldsymbol{\theta}_{k+\frac{1}{2}} = \boldsymbol{\theta}_k - \gamma \mathbf{S}_k$$
$$\mathbf{S}_{k+1} = \gamma \mathbf{S}_k + \mu_k \nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_{k+\frac{1}{2}})$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{S}_{k+1}$$

Improvements of SGD: AdaGrad

Idea: Scale step according to size of weights (relation to prior-conditioning in SGD)

Iteration:

$$\mathbf{D}_{k+1} = \boldsymbol{\theta}_k^2 + \mathbf{D}_k$$
$$\mathbf{S}_{k+1} = \mu_k \mathrm{diag}(\mathbf{D}_{k+1})^{-1} \nabla F_{\mathcal{S}_k}(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{S}_{k+1}$$

Discussion: Stochastic Approximation

General Comments:

- Lots of theory for convex problems
- Recall: SGD is not the best tool for most convex problems (see example of least-squares)
- Require very careful tuning

SGD in deep learning:

- currently the main workhorse (DNN $\rightsquigarrow$ nonconvex optimization)
- why it works? mostly open but some relation to Langevin flow (we also have a few ideas)
- observed to regularize problems (theory for quadratic case)
- potentially possible to prove global optimality?

Practical Hint: Data Preprocessing

Some practical tips

- Remove the mean of the data
- Scale it to be "reasonable" scale
- Data augmentation
- Some other (domain specific) data transforms (optical flow for motion?)

Regularization for Network Weights

- Note that there are many more degrees of freedom.
- Need to add regularization for $\mathbf{K}$
- $\mathbf{K}$ Generally, $\mathbf{K}$ is not "physical" - difficult to choose reasonable regularization.

The obvious choice: Tikhonov

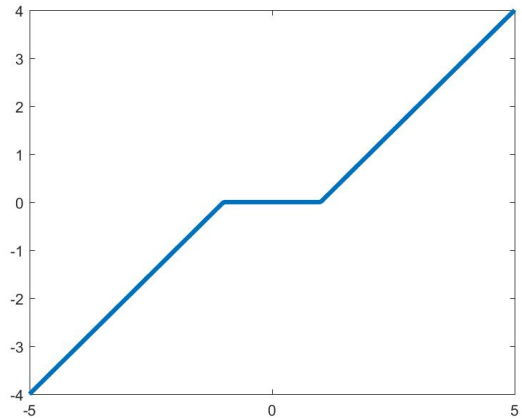
$$R(\mathbf{K}) = \frac{1}{2}\|\mathbf{K}\|_F^2$$

(also called weight decay)

Learning the weights - Regularization

More recent, demand that $\mathbf{K}$ is sparse

$$R(\mathbf{K}) = \|\mathrm{vec}(\mathbf{K})\|_1 = \sum_{ij} |\mathbf{K}_{ij}|$$



Implementation through soft-thresholding.
After each steepest descent iteration set

$$\mathbf{K} = \mathrm{softThresh}(\mathbf{K})$$

Obtain sparse matrices $\mathbf{K}$ that retain only necessary entries

Test Problems

Before going to real data, let us try the *inverse crime*.
Generate data

```
n  = 500; nf = 50; nc = 10; m  = 40;
Wtrue = randn(nc,m);
Ktrue = randn(m,nf);
btrue = .1;

Y     = randn(nf,n);
Cobs  = exp(Wtrue*singleLayer(Ktrue,btrue,Y));
Cobs  = Cobs./sum(Cobs,1);
```

Goal: Reconstruct `Wtrue, Ktrue, btrue`!

Other cheap test problems: `PeaksClassification`, `PeaksRegression`, `CircleClassification`.

Σ : Single-Layer Neural Networks

$$\min_{\mathbf{K},\mathbf{W},\mathbf{b}} E(\mathbf{W}\sigma(\mathbf{K}\mathbf{Y} + \mathbf{b}\mathbf{e}_n^\top), \mathbf{C}_{\text{obs}}) + \lambda R(\mathbf{W}, \mathbf{K}, \mathbf{b})$$

- transform data, increase dimension $\rightsquigarrow$ approximation power
- Extreme Learning Machines: random nonlinear feature extractor
- More common to train $\mathbf{K}, \mathbf{b}, \mathbf{W}$
- Training problem is non-convex and stochastic
- SAA methods: Pick large sample and use deterministic tools (easy to parallelize, fast convergence if done right, but can be trapped in local minima)
- SA methods: small sample and random steps (easy to code, difficult to parallelize, need to choose hyper parameter)

References

[1] D. P. Bertsekas. Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey. *arXiv preprint [cs.SY 1507.01030v1]*, 2015.

[2] L. Bottou. Stochastic gradient descent tricks. *Neural networks: Tricks of the trade*, 2012.

[3] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *arXiv preprint [stat.ML] (1606.04838v1)*, 2016.

[4] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[5] G. Golub and V. Pereyra. Separable nonlinear least squares: the variable projection method and its applications. *Inverse Problems*, 19:R1–R26, 2003.

[6] G. H. Golub and V. Pereyra. The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate. *SIAM Journal on Numerical Analysis*, 10(2):413–432, 1973.

[7] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[8] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1-3):489–501, Dec. 2006.

[9] E. Newman, L. Ruthotto, J. Hart, and B. van Bloemen Waanders. Train Like a (Var)Pro: Efficient Training of Neural Networks with Variable Projection. *arXiv.org*, July 2020.

References (cont.)

[10] D. P. O'Leary and B. W. Rust. Variable projection for nonlinear least squares problems. *Computational Optimization and Applications. An International Journal*, 54(3):579–593, 2013.

[11] H. Robbins and S. Monro. A Stochastic Approximation Method. *The annals of mathematical statistics*, 22(3):400–407, 1951.