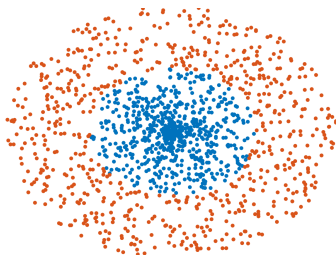


One-Layer Neural Networks

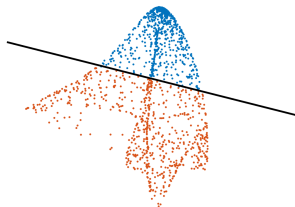
Numerical Methods for Deep Learning

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 the Weights

Assume that the number of examples, n , is very large.
Using random weights, \mathbf{K} might need to be very large to fit training data.

Solution may not generalize well to test data.

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

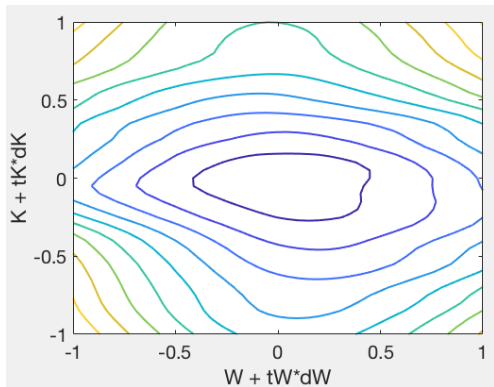
$$\min_{\mathbf{K}, \mathbf{W}, b} E(\mathbf{W}\sigma(\mathbf{K}\mathbf{Y} + b), \mathbf{C}^{\text{obs}}) + \lambda R(\mathbf{W}, \mathbf{K}, b)$$

About this optimization problem:

- ▶ more unknowns $\mathbf{K} \in \mathbb{R}^{m \times n_f}$, $\mathbf{W} \in \mathbb{R}^{n_c \times m}$, $b \in \mathbb{R}$
- ▶ non-convex problem \leadsto local minima, careful initialization
- ▶ need to compute derivatives w.r.t. \mathbf{K}, b

Non-Convexity

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

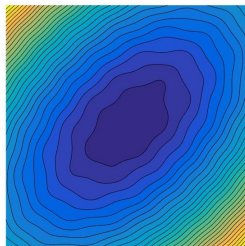


(see E08SingleLayerNN.m)

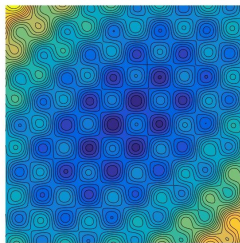
Expect worse when number of layers grows!

Training the Neural Network

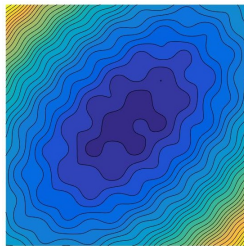
- ▶ If non-convexity is not “too bad” can use standard gradient based methods
- ▶ If non-convexity is “ugly” need to modify standard methods (stochastic kick)
- ▶ If non-convexity is “bad” need global optimization techniques



good



bad



ugly

Recap: Differentiating Linear Algebra Expressions

Easy ones:

$$F_1(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top \mathbf{y}$$

$$\mathbf{J}_x F_1(\mathbf{x}, \mathbf{y}) = \mathbf{y}^\top$$

$$F_2(\mathbf{A}, \mathbf{x}) = \mathbf{A}\mathbf{x}$$

$$\mathbf{J}_x F_2(\mathbf{x}, \mathbf{y}) = \mathbf{A}$$

How about

$$F_3(\mathbf{A}, \mathbf{X}) = \mathbf{A}\mathbf{X} \quad \mathbf{J}_{\text{vec}(\mathbf{X})} F_3 = ???$$

Recall that

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

Therefore:

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

Efficient mat-vec: $\mathbf{J}_{\text{vec}(\mathbf{X})} F \mathbf{v} = \text{vec}(\mathbf{A} \text{ mat}(\mathbf{v}))$

Training Single Layer Neural Network

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

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

Agenda:

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

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

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

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

Computing Jacobians

$$\mathbf{Z} = \sigma(\mathbf{KY} + b)$$

Recall that σ is applied element-wise.

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

Efficient way to get matrix vector products

$$\begin{aligned}\mathbf{J}_{\text{vec}(\mathbf{K})}\mathbf{Z}\mathbf{v} &= \text{diag}(\sigma'(\mathbf{KY} + b))(\mathbf{Y}^\top \otimes \mathbf{I})\mathbf{v} \\ &= \text{vec}(\sigma'(\mathbf{KY} + b) \odot (\text{mat}(\mathbf{v})\mathbf{Y}))\end{aligned}$$

And for transpose get

$$\begin{aligned}(\mathbf{J}_{\text{vec}(\mathbf{K})}\mathbf{Z})^\top \mathbf{u} &= (\mathbf{Y} \otimes \mathbf{I})\text{diag}(\sigma'(\mathbf{KY} + b))\mathbf{u} \\ &= \text{vec}(\sigma'(\mathbf{KY} + b) \odot \text{mat}(\mathbf{u})\mathbf{Y}^\top)\end{aligned}$$

Class Problems: Derivatives of Single Layer

Derivations:

1. Compute $\mathbf{J}_b \mathbf{Z} \mathbf{v}$ and $(\mathbf{J}_b \mathbf{Z})^\top \mathbf{u}$
2. Compute $\mathbf{J}_{\text{vec}(\mathbf{Y})} \mathbf{Z} \mathbf{v}$ and $(\mathbf{J}_{\text{vec}(\mathbf{Y})} \mathbf{Z})^\top \mathbf{u}$

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

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

Putting Things Together

Implement loss function of single-layer NN

$$E(\mathbf{K}, b, \mathbf{W}) \stackrel{\text{def}}{=} E(\mathbf{C}, \mathbf{Z}, \mathbf{W}), \quad \mathbf{Z} = \sigma(\mathbf{K}\mathbf{Y} + b)$$

```
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. $\nabla_{\mathbf{W}} E = \nabla_{\mathbf{S}} E(\mathbf{S})\mathbf{Y}$

Test Problem

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!

Gauss-Newton Method

Goal: Accelerate convergence by using curvature information.

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

Denoting $\mathbf{J}_{\mathbf{K}} \mathbf{Z} = \nabla_{\mathbf{K}} \sigma(\mathbf{K} \mathbf{Y} + b)^{\top}$ this means that Hessian is

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

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)

inexact Hessian + inexact solve: add line search!

Experiment: Adversarial Example

Suppose you have trained your network $\rightsquigarrow \mathbf{K}, b, \mathbf{W}$ so that validation loss is low. This means that for most examples \mathbf{y} ,

$$\mathbf{W}\sigma(\mathbf{K}\mathbf{y} + b) \approx \mathbf{c}.$$

An adversary might try to fool this classifier by adding a small perturbation \mathbf{d} to the example to achieve a desired label $\hat{\mathbf{c}}$.

Formulate as optimization problem

$$\min_{\mathbf{d}} E(\mathbf{W}\sigma(\mathbf{K}(\mathbf{y} + \mathbf{d}) + b), \hat{\mathbf{c}})$$

- ▶ setup objective function
- ▶ think about constraints, regularization