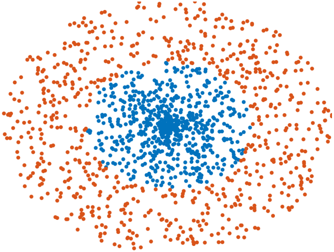
Our First Neural Network

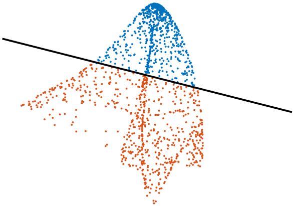
Numerical Methods for Deep Learning

Motivation: Nonlinear Models

In general, impossible to find a linear separator between points



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 $\mathbf{W}$)

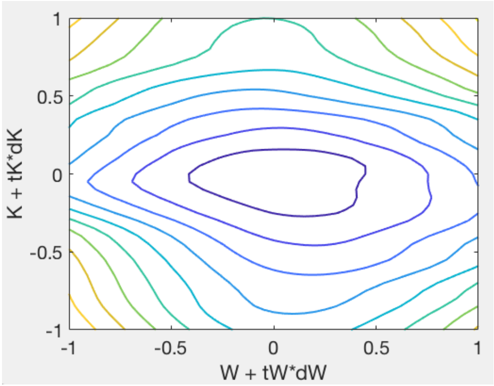
$$\min_{\mathbf{K},\mathbf{W},b} E(\sigma(\mathbf{Y}\mathbf{K}+b)\mathbf{W}, \mathbf{C}^{\mathrm{obs}}) + \alpha R(\mathbf{W},\mathbf{K},\mathbf{b})$$

About this optimization problem:

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

Non-Convexity

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

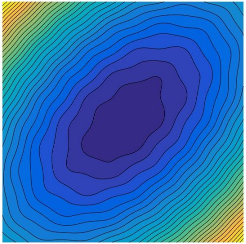


Test this using `exSingleLayerNN.m`

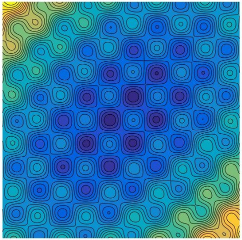
Expect worse when number of layers grows!

Training the NN

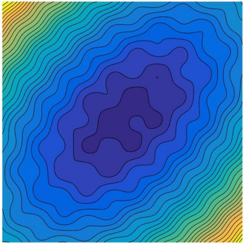
- If non-convexity is not “too bad” can use standard gradient based methods
- If non-convexity is “ugly” need to modify standard methods (stochastic kick)
- If non-convexity is “bad” need global optimization techniques



good



bad



ugly

Recap: Differentiating Linear Algebra Expressions

Easy ones:

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

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

How about

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

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:

$$\nabla_{\mathrm{vec}(\mathbf{X})} F_3(\mathbf{A}, \mathbf{X}) = \mathbf{I} \otimes \mathbf{A}$$

Efficient mat-vec: $\nabla_{\mathrm{vec}(\mathbf{X})} F \mathbf{v} = \mathrm{vec}(\mathbf{A}\ \mathrm{mat}(\mathbf{v}))$

Training Single Layer Neural Network

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

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

Agenda:

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

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

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

Computing Jacobians

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

Recall that σ is applied element-wise.

$$\mathbf{J}_{\mathrm{vec}(\mathbf{K})} = \mathrm{diag}(\sigma'(\mathbf{YK} + b))(\mathbf{I} \otimes \mathbf{Y})$$

Need only matrix vector products

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

Class Problems: Derivatives of Single Layer

Derivations:

1. Find efficient way to compute $\mathbf{J}_{\mathrm{vec}(\mathbf{K})}^\top \mathbf{u}$
2. Compute $\mathbf{J}_b v$ and $\mathbf{J}_b^\top \mathbf{u}$
3. Compute $\mathbf{J}_{\mathrm{vec}(\mathbf{Y})} \mathbf{v}$ and $\mathbf{J}_{\mathrm{vec}(\mathbf{Y})}^\top \mathbf{u}$

Coding:

```
function[Z,JKt,Jbt,JYt,JK,Jb,JY] = singleLayer(K,b,Y)
% Returns Z = sigma(Y*K+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

$$E(\mathbf{K}, b, \mathbf{W}) = E(\mathbf{ZW}, \mathbf{C}), \quad \mathbf{Z} = \sigma(\mathbf{YK} + b)$$

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

Use

1. $\nabla_{\mathbf{Z}} E = \nabla_{\mathbf{S}}\, E(\mathbf{S})\mathbf{W}^\top, \quad \mathbf{S} = \mathbf{ZW}$
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. $\nabla_{\mathbf{W}} E = \mathbf{Y}^\top \nabla_{\mathbf{S}}\, E(\mathbf{S})$

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(m,nc);
Ktrue = randn(nf,m);
btrue = .1;

Y     = randn(n,nf);
Cobs  = exp(singleLayer(Ktrue,btrue,Y)*Wtrue);
Cobs  = Cobs./sum(Cobs,2);
```

Goal: Reconstruct it!

Gauss-Newton Method

Goal: Accelerate convergence by using curvature information.
Recall

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

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

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

First term is always 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)

add line search to be safe!

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 $\mathbf{y}$,

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

An adversary might want 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

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

- setup objective function
- think about constraints, regularization