CS 559 Machine Learning

Neural Networks

Yue Ning
Department of Computer Science
Stevens Institute of Technology

Plan for today

Summary and Review

Neural Networks

Learning Outcome

- Implement a neural network for classification
- Implement Adagrad

Summary of supervised learning

$$\underbrace{\mathbf{w}^T\mathbf{x} + b}_{\text{score}}$$

	Classification	Regression
Predictor	sign	score
Relate to y	margin (score, y)	residual (score, y)
Loss	zero-one, hinge, logistic	squared, absolute deviation
Algorithm	GD	GD

Review: Optimization problem

Minimize Training Loss

$$\min_{\mathbf{w} \in \mathbb{R}^d} \text{TrainLoss}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} L(\mathbf{x}_n, y_n, \mathbf{w})$$

Review: Optimization problem

Gradient Descent (GD)

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

$\nabla_{\mathbf{w}}$TrainLoss denotes the gradient of the (average) total training loss with respect to $\mathbf{w}$

Stochastic Gradient Descent (SGD)

For each $(x, y) \in D_{train}$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{L}(x, y, \mathbf{w})$$

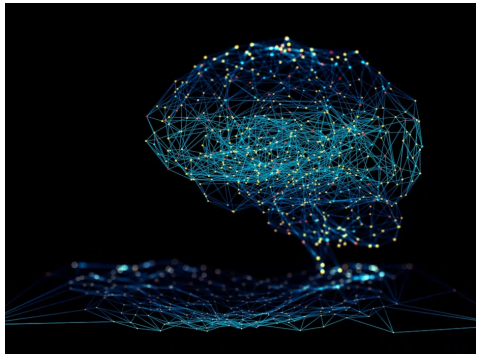
$\nabla_{\mathbf{w}} L$ denotes the gradient of one example loss with respect to w

Notation clarification

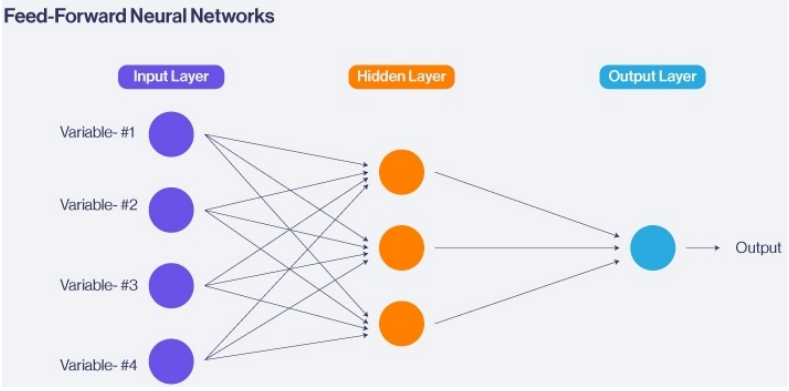
- $\mathbf{w}, \mathbf{x}$: bold letters are usually vectors in dimension d
- w_i: the i-th value in vector $\mathbf{w}$
- $\mathbf{x}_n$: the n-th example in the training data set, its corresponding target value: y_n
- x_i: feature i in current example $\mathbf{x}$
- x_{ni}: feature i in the n-th example
- V, W: capital letters usually denote matrices.

What is Neural Network?

- Often associated with biological devices (brains), electronic devices, or network diagrams;



- But the best conceptualization for this presentation is none of these: think of a neural network as a mathematical function.



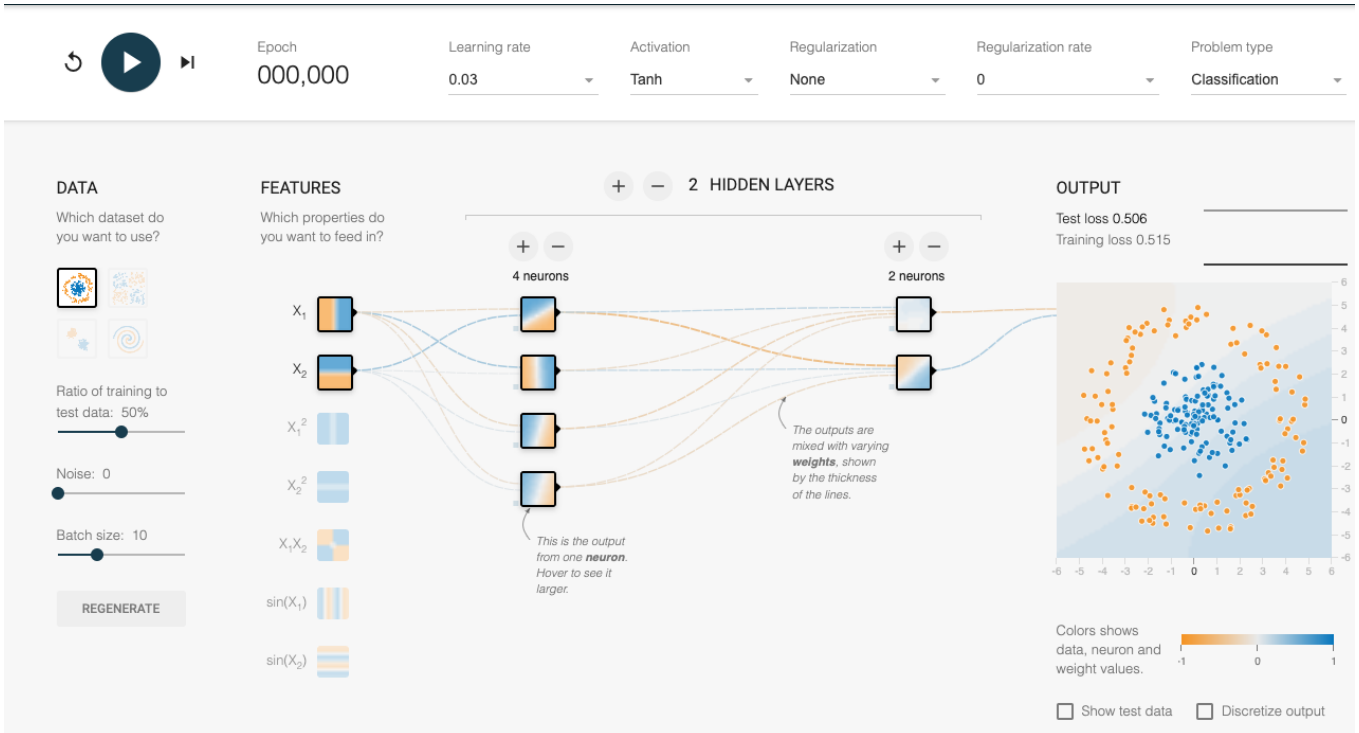
The pros of Neural Network

- Successfully used on a variety of domains:
  - computer vision
  - speech recognition
  - gaming
- Can provide solutions to very complex and nonlinear problems;

- If provided with sufficient amount of data, can solve classification and forecasting problems accurately and easily
- Once trained, prediction is fast;

Playground

Click Neural Network Playground



Credits: Daniel Smilkov and Shan Carter

Motivation

- Perceptron: limitations;
- Feedforward networks and Backpropagation;

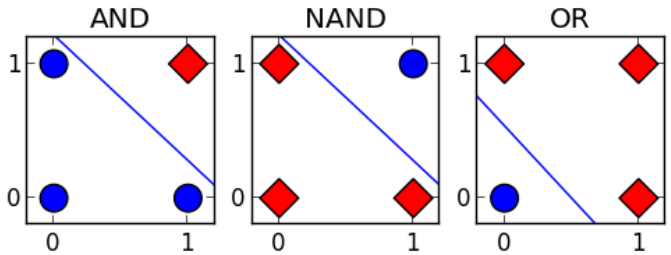


Figure: Linearly Separable Functions

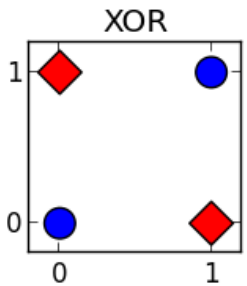
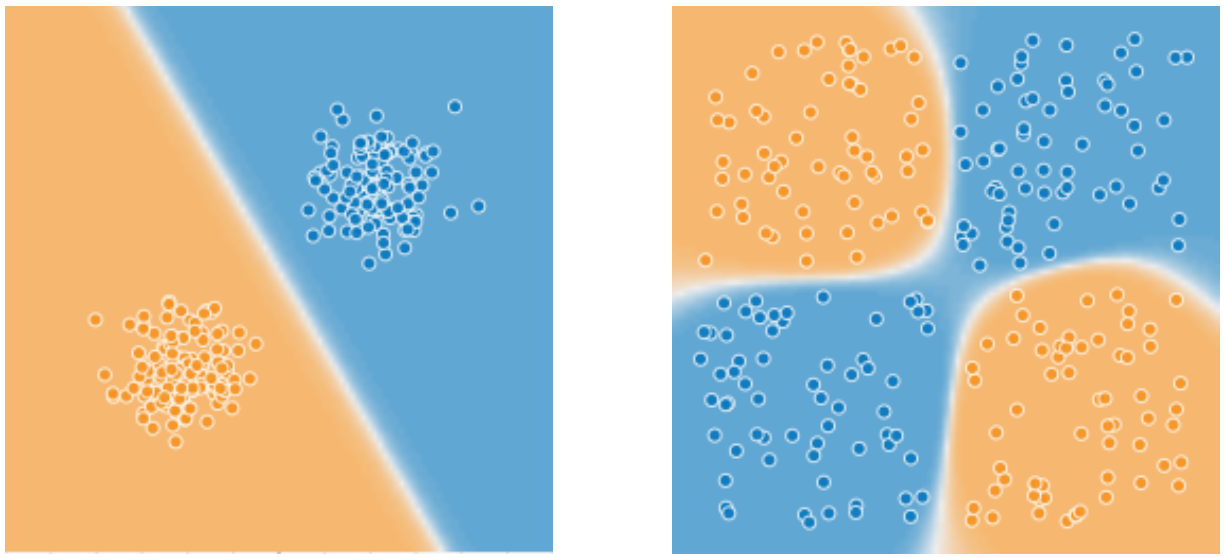


Figure: Non-Linearly Separable Functions

Motivation

- Allow to learn non linearly separable transformations from input to output;



(a) Linear Classifier (b) Nonlinear Classifier

- A single hidden layer allows to compute any input/output transformation;

Motivation - Example

Toy example: predicting car collision

Input: position of two oncoming cars $\mathbf{x} = [x_1, x_2]^T$
Output: whether safe ($y = +1$) or collide ($y = -1$)

- True function: safe if cars are far (at least 1)

$$y = \text{sign}(|x_1 - x_2| - 1)$$

- Examples:

$\mathbf{x}$	y
$[1, 3]^T$	$+1$
$[3, 1]^T$	$+1$
$[1, 0.5]^T$	-1

Decomposing the problem

- Test if car 1 is far right of car 2:

$$h_1 = \mathbb{I}(x_1 - x_2 \geq 1)$$

- Test if car 2 is far right of car 1:

$$h_2 = \mathbb{I}(x_2 - x_1 \geq 1)$$

- Safe if at least one is true:

$$y = \text{sign}(h1 + h2)$$

$\mathbf{x}$	h_1	h_2	y
$[1, 3]^T$	0	1	+1
$[3, 1]^T$	1	0	+1
$[1, 0.5]^T$	0	0	-1

Learning Strategy

- Define: $\mathbf{x} = [1, x_1, x_2]^T$
- Intermediate hidden subproblems:

$$h_1 = \mathbb{I}(\mathbf{v}_1{}^T \mathbf{x} \geq 0), \mathbf{v}_1 = [-1, +1, -1]^T$$

$$h_2 = \mathbb{I}(\mathbf{v}_2{}^T \mathbf{x} \geq 0), \mathbf{v}_2 = [-1, -1, +1]^T$$

- Final prediction:

$$f_{V,\mathbf{w}}(x) = \text{sign}(w_1 h_1 + w_2 h_2), \mathbf{w} = [w_1, w_2] = [1, 1]$$

Key idea: joint learning

Goal: learn both hidden subproblems and combination weights

Gradients

- Problem: gradient of h_1 with respect to $\mathbf{v}_1$ is 0

$$h_1 = \mathbb{I}(\mathbf{v}_1^T \mathbf{x} \geq 0)$$

- The logistic function maps $(-\infty, +\infty)$ to $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

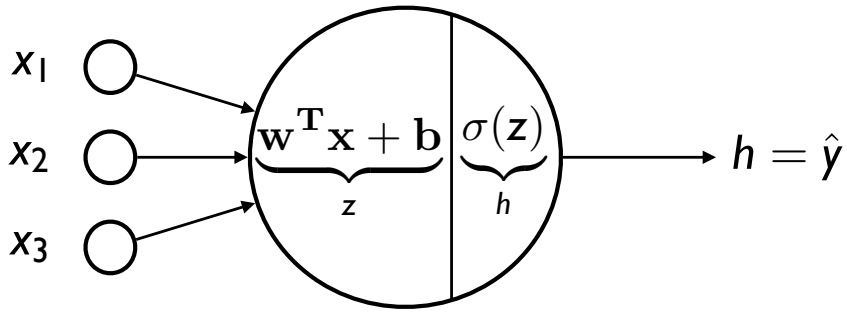
Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

- Solution: $h_j = \sigma(\mathbf{v}_j{}^T \mathbf{x})$

No Hidden Units: Logistic Regression

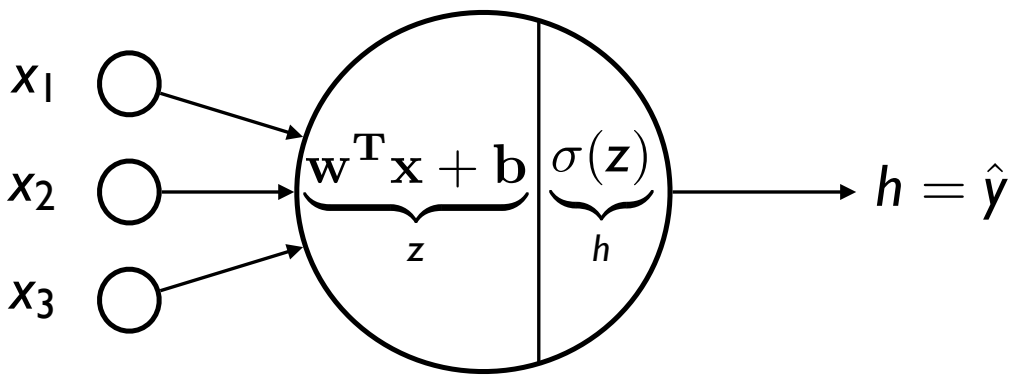
- Sigmoid activation function:



- Output score: sigmoid function $\hat{y} = \sigma(\mathbf{w}^T\mathbf{x} + b)$

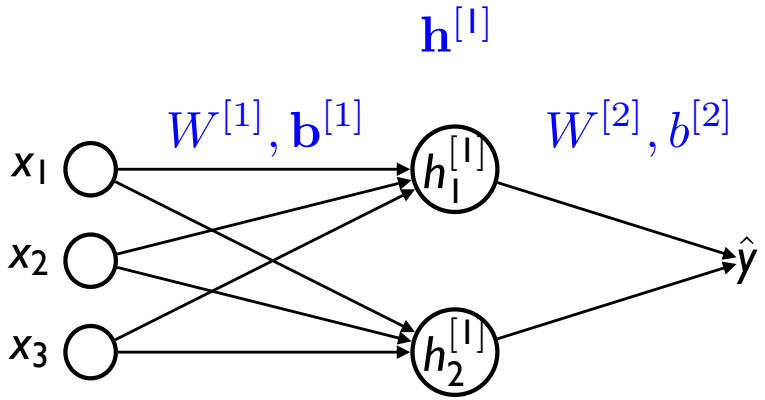
Learning Algorithm: No hidden units

- Logistic loss: $J(\mathbf{x}, y, \mathbf{w}) = -y \log(\hat{y}) - (1-y)\log(1-\hat{y})$ where $\hat{y} = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$
- $\frac{\partial \hat{y}}{\partial \mathbf{w}} = \hat{y}(1-\hat{y})\mathbf{x}$ element-wise: $\frac{\partial \hat{y}}{\partial w_i} = \hat{y}(1-\hat{y})x_i$
- $\frac{\partial J}{\partial \mathbf{w}} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \mathbf{w}} = (\hat{y}-y)\mathbf{x}$, element-wise: $\frac{\partial J}{\partial w_i} = (\hat{y}-y)x_i$
- Applying gradient descent (η is the learning rate):
  - Element (feature) operation: $w_i^{t+1} = w_i^t - \eta\frac{\partial J}{\partial w_i}$
  - Vector operation: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta\frac{\partial J}{\partial \mathbf{w}}$



Neural Networks: one hidden layer

- One hidden layer:



- Hidden layer representation

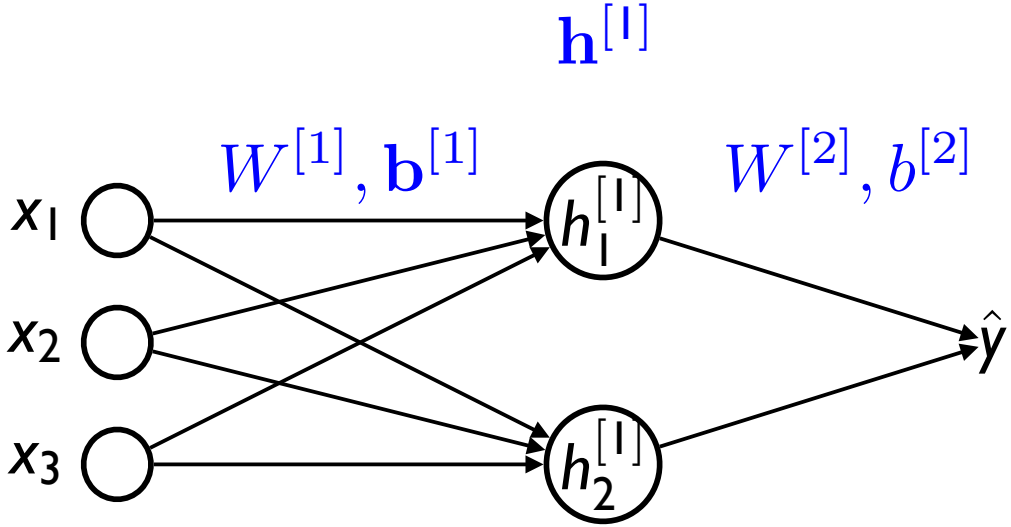
$$z_1^{[1]} = \mathbf{w}_1^{[1]T}\mathbf{x} + b_1^{[1]}, \qquad h_1^{[1]} = g(z_1^{[1]})$$
$$z_2^{[1]} = \mathbf{w}_2^{[1]T}\mathbf{x} + b_2^{[1]}, \qquad h_2^{[1]} = g(z_2^{[1]})$$

Vectorization:

$$\mathbf{z}^{[1]} = \underbrace{W^{[1]}}_{2\times 3}\underbrace{\mathbf{x}}_{3\times 1} + \underbrace{\mathbf{b}^{[1]}}_{2\times 1}$$
$$\mathbf{h}^{[1]} = g(\mathbf{z}^{[1]})$$

Neural Networks: one hidden layer

- One hidden layer:



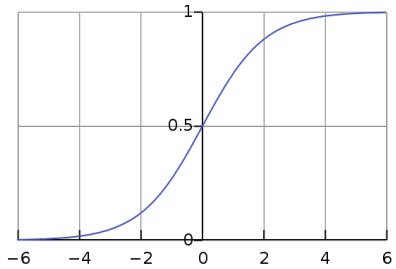
- Output layer

$$z^{[2]} = \underbrace{W^{[2]}}_{1\times 2}\underbrace{\mathbf{h}^{[1]}}_{2\times 1} + b^{[2]}$$

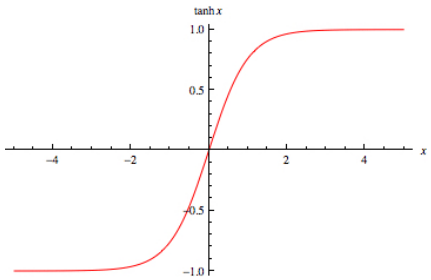
$$\hat{y} = g(z^{[2]})$$

Activation Function g()

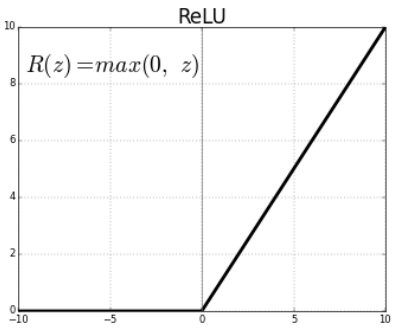
Sigmoid: $g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$



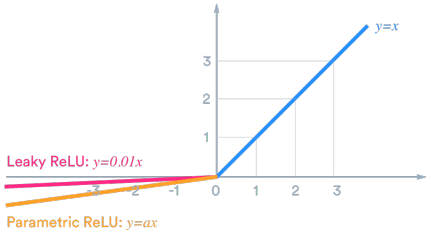
tanh: $g(x) = 2\sigma(2x) - 1$



ReLU: $g(x) = \max(0, x)$



Leaky ReLU: $g(x) = \max(\alpha x, x)$

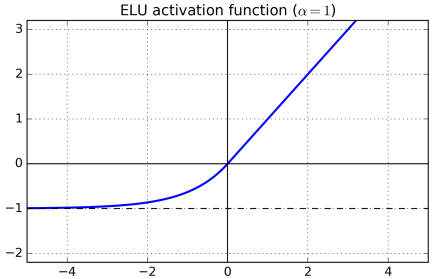


Maxout

$$\max(\mathbf{w}_1^T \mathbf{x} + b_1, \mathbf{w}_2^T \mathbf{x} + b_2)$$

ELU:

$$g(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Neural Networks: one hidden layer

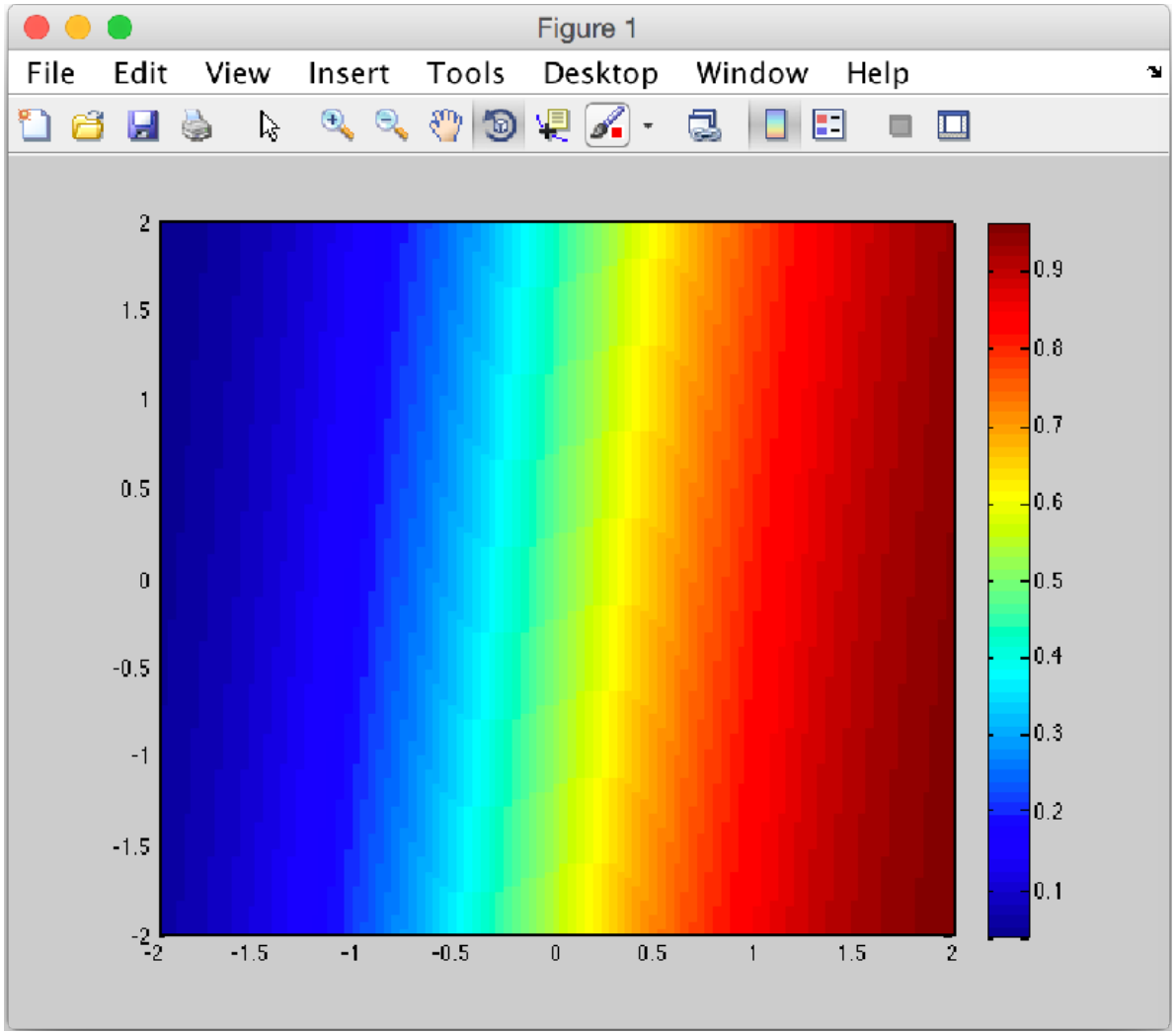
- Think of intermediate hidden units as learned features of a linear predictor.
- Feature learning: manually specified features:

$$\mathbf{x}$$

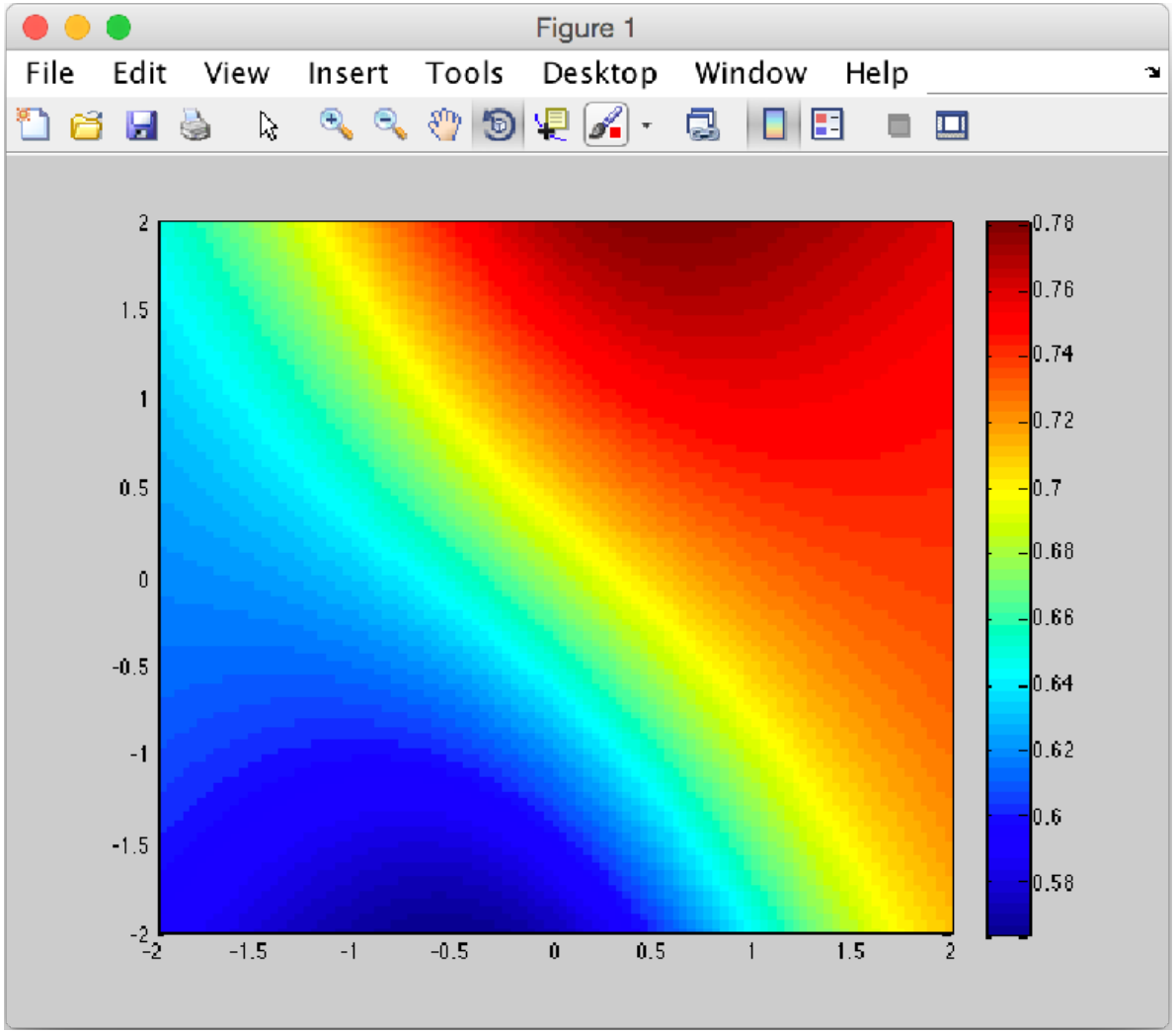
automatically learned features:

$$h(\mathbf{x}) = [h_1(\mathbf{x}), ..., h_k(\mathbf{x})]$$

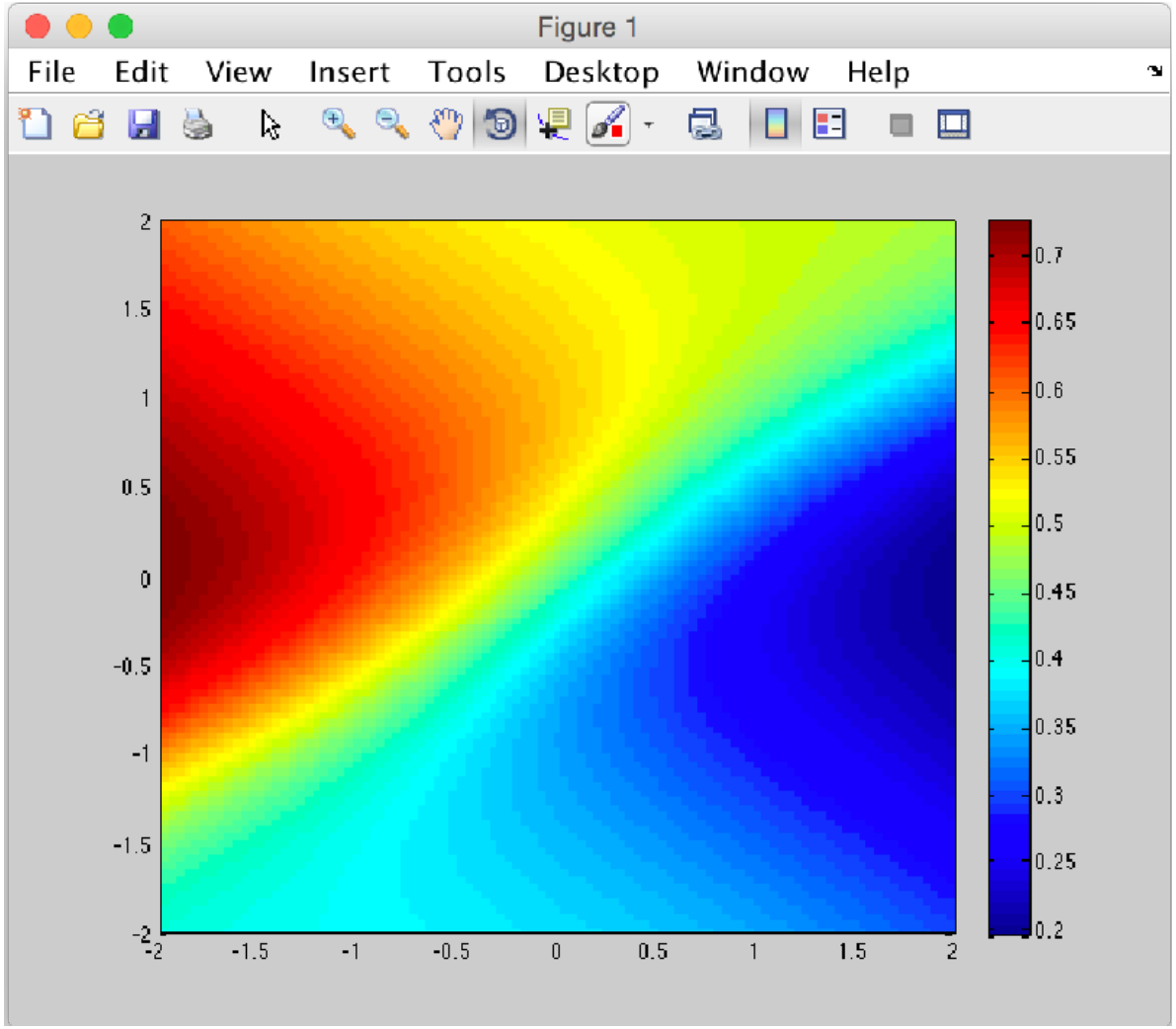
Decision Boundary: Logistic Regression



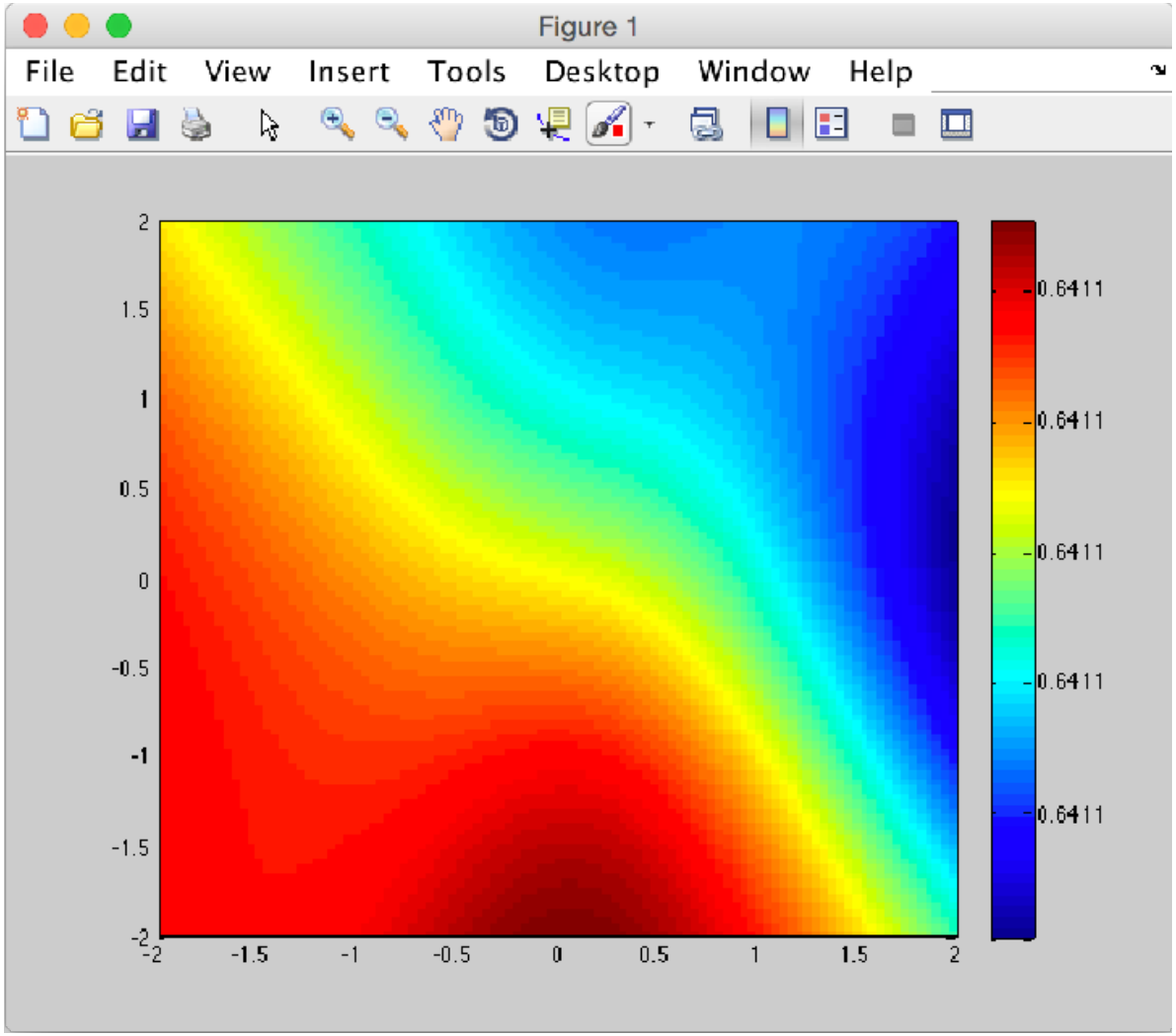
Decision Boundary: 2 hidden layers, 2 hidden units



Decision Boundary: 2 hidden layers, 3 hidden units



Decision Boundary: 10 hidden layers, 5 hidden units



Loss Minimization

- Optimization problem

$$\text{TrainLoss}(\mathbf{W}, \mathbf{b}) = \frac{1}{|\mathcal{D}_{train}|} \sum_{(\mathbf{x},y)\in\mathcal{D}_{train}} \text{Loss}(\mathbf{x}, y, \mathbf{W}, \mathbf{b})$$

$$J = \text{Loss}(\mathbf{x}, y, \mathbf{W}, \mathbf{b}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

$$\hat{y} = h^{[2]}$$

- Goal: compute gradient

$$\nabla_{\mathbf{W},\mathbf{b}} J(\mathbf{W}, \mathbf{b})$$

Stochastic Gradient Descent

- Goal: compute gradient

$$\nabla_{\mathbf{W},\mathbf{b}} J(\mathbf{W}, \mathbf{b})$$

- Repeat, given a sample $(\mathbf{x}^{(i)}, y^{(i)})$:

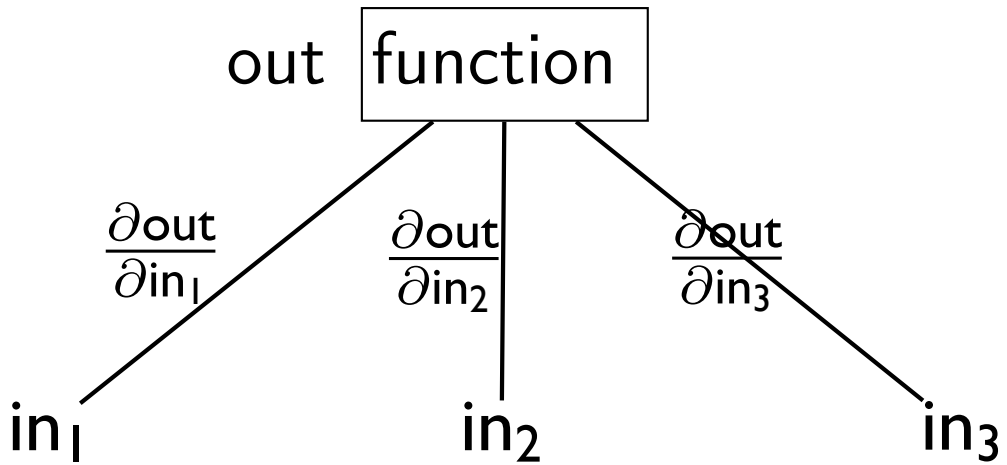
$$dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}, \quad W^{[1]} = W^{[1]} - \eta dW^{[1]}$$
$$d\mathbf{b}^{[1]} = \frac{\partial J}{\partial \mathbf{b}^{[1]}}, \quad \mathbf{b}^{[1]} = \mathbf{b}^{[1]} - \eta d\mathbf{b}^{[1]}$$
$$dW^{[2]} = \frac{\partial J}{\partial W^{[2]}}, \quad W^{[2]} = W^{[2]} - \eta dW^{[2]}$$
$$db^{[2]} = \frac{\partial J}{\partial b^{[2]}}, \quad b^{[2]} = b^{[2]} - \eta db^{[2]}$$

Approach

- Mathematical: just grind through the chain rule
- Next: visualize the computation using a computation graph
- Advantages:
  - Avoid long equations
  - Reveal structure of computations (modularity, efficiency, dependencies)

Functions as boxes

- Partial derivatives (gradients): how much does the output change if an input changes?



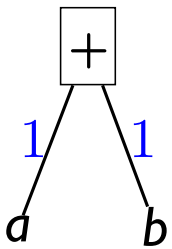
- Example:

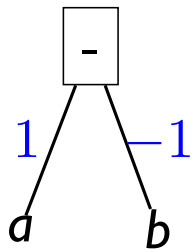
$$\text{out} = 2\text{in}_1 + \text{in}_2\text{in}_3$$

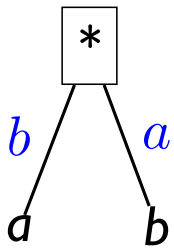
$$2\text{in}_1 + (\text{in}_2 + \epsilon)\text{in}_3 = \text{out} + \epsilon\text{in}_3$$

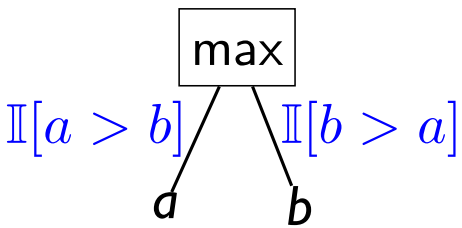
- Partial derivatives (gradients): a measure of sensitivity

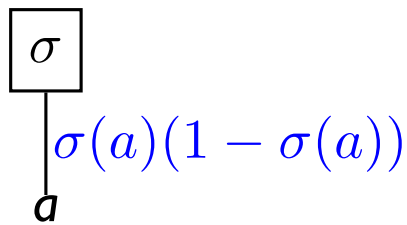
Basic building blocks



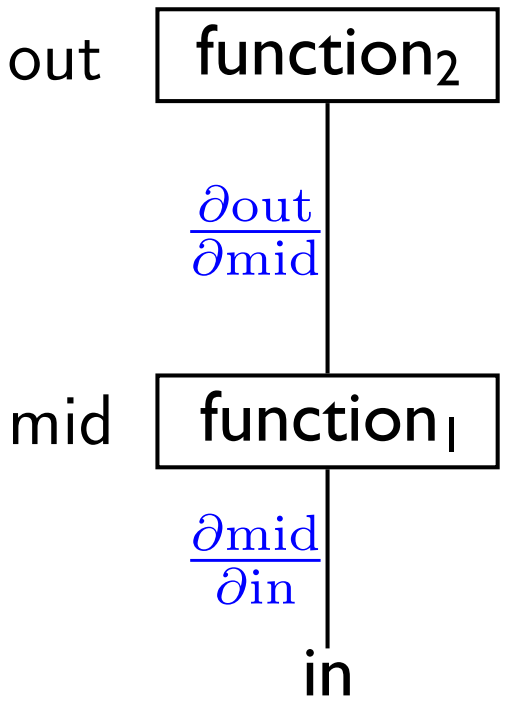








Composing Functions



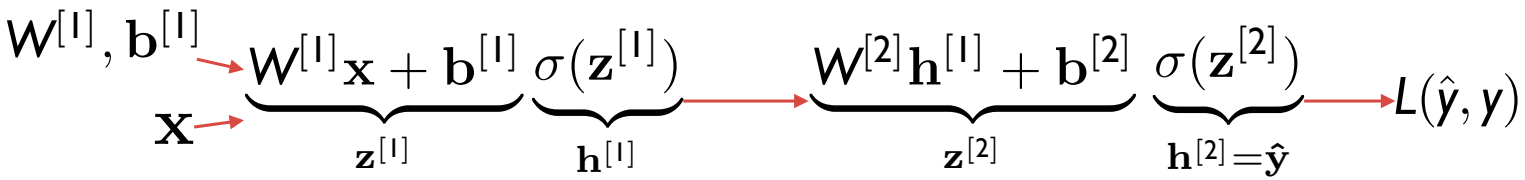
Chain rule

$$\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$$

Back propagation

- Goal: learn the weights so that the loss (error) is minimized.
- It provides an efficient procedure to compute derivatives.
- For a fixed sample $(\mathbf{x}, y)$, we want to learn $\hat{y} = f(\mathbf{x})$
- Negative Log-likelihood over input $\mathbf{x}_n$ is
  $L_n = -y_n \log(\hat{y}_n) - (1 - y_n) \log(1 - \hat{y}_n)$
- Total error of training examples is $E = \sum_n L_n$

Back propagation



Loss function : $L = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$

$$d\mathbf{z}^{[1]} = \frac{\partial L}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial \mathbf{z}^{[1]}} = {W^{[2]}}^T dz^{[2]} \circ \mathbf{h}^{[1]} \circ (1 - \mathbf{h}^{[1]})$$

$$dW^{[1]} = d\mathbf{z}^{[1]} \mathbf{x}^T$$

$$d\mathbf{b}^{[1]} = d\mathbf{z}^{[1]}$$

$$dz^{[2]} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{[2]}} = \hat{y} - y$$

$$dW^{[2]} = dz^{[2]} {\mathbf{h}^{[1]}}^T$$

$$db^{[2]} = dz^{[2]}$$

Summary: computing derivatives

Forward propagation:

$$\mathbf{z}^{[1]} = W^{[1]}\mathbf{x} + b^{[1]}$$
$$\mathbf{h}^{[1]} = \sigma(\mathbf{z}^{[1]})$$
$$z^{[2]} = W^{[2]}\mathbf{h}^{[1]} + b^{[2]}$$
$$\hat{y} = h^{[2]} = \sigma(z^{[2]})$$

Backpropagation

$$dz^{[2]} = h^{[2]} - y$$
$$dW^{[2]} = dz^{[2]}\mathbf{h}^{[1]\,T}$$
$$db^{[2]} = dz^{[2]}$$
$$d\mathbf{z}^{[1]} = W^{[2]\,T} dz^{[2]} \circ \mathbf{h}^{[1]} \circ (1 - \mathbf{h}^{[1]})$$
$$dW^{[1]} = d\mathbf{z}^{[1]}\mathbf{x}^{T}$$
$$d\mathbf{b}^{[1]} = d\mathbf{z}^{[1]}$$

Vectorizing across multiple examples

forward pass

for $n = 1$ to N:

$$\mathbf{z}_n^{[1]} = W^{[1]}\mathbf{x}_n + b^{[1]}$$
$$\mathbf{h}_n^{[1]} = \sigma(\mathbf{z}_n^{[1]})$$
$$z_n^{[2]} = W^{[2]}\mathbf{h}_n^{[1]} + b^{[2]}$$
$$h_n^{[2]} = \sigma(z_n^{[2]})$$

Vectorizing:

$$Z^{[1]} = W^{[1]}X + \mathbf{b}^{[1]}$$
$$H^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}H^{[1]} + b^{[2]}$$
$$H^{[2]} = \sigma(Z^{[2]})$$

Vectorizing across multiple examples

backpropagation

Given one data point:

$$dz^{[2]} = h^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} \mathbf{h}^{[1]\,T}$$

$$db^{[2]} = dz^{[2]}$$

$$d\mathbf{z}^{[1]} = dz^{[2]} W^{[2]\,T} \circ \mathbf{h}^{[1]} \circ (1 - \mathbf{h}^{[1]})$$

$$dW^{[1]} = d\mathbf{z}^{[1]} \mathbf{x}^{T}$$

$$d\mathbf{b}^{[1]} = d\mathbf{z}^{[1]}$$

Vectorizing:

$$dZ^{[2]} = H^{[2]} - \mathbf{y}$$

$$dW^{[2]} = \frac{1}{N} dZ^{[2]} H^{[1]\,T}$$

$$db^{[2]} = \frac{1}{N} \sum dZ^{[2]}$$

$$dZ^{[1]} = W^{[2]\,T} dZ^{[2]} \circ H^{[1]} \circ (1 - H^{[1]})$$

$$dW^{[1]} = \frac{1}{N} dZ^{[1]} X^{T}$$

$$d\mathbf{b}^{[1]} = \frac{1}{N} \sum d\mathbf{z}^{[1]}$$

STEVENS

Break

Adaptive learning rates

- Popular and simple idea: reduce the learning rate by some factor every few epochs.
  - At the beginning, we are far from the destination, so we use larger learning rate
  - After several epochs, we are close to the destination, so we reduce the learning rate
  - E.g. $1/t$ decay: $\eta^t = \frac{\eta}{\sqrt{t+1}}$
- Learning rate cannot be one-size-fits-all
  - Giving different parameters different learning rates.

Adaptive learning rates

$$\eta^t = \frac{\eta}{\sqrt{t+1}}, g^t = \frac{\partial L(w^t)}{\partial w}$$

- Divide the learning rate of each parameter by the root mean square of its previous derivatives.
- Vanilla gradient descent:

$$w^{t+1} \leftarrow w^t - \eta^t g^t$$

- Adagrad:

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t$$

σ^t: root mean square of the previous derivatives of w

Adagrad

$$
\begin{aligned}
&\text{t=1: } w^{(1)} \leftarrow w^{(0)} - \frac{\eta^{(0)}}{\sigma^{(0)}} g^{(0)} && \sigma^{(0)} = \sqrt{(g^{(0)})^2 + \epsilon} \\
&\text{t=2: } w^{(2)} \leftarrow w^{(1)} - \frac{\eta^{(1)}}{\sigma^{(1)}} g^{(1)} && \sigma^{(1)} = \sqrt{\frac{1}{2}[(g^{(0)})^2 + (g^{(1)})^2] + \epsilon} \\
&\text{t=3: } w^{(3)} \leftarrow w^{(2)} - \frac{\eta^{(2)}}{\sigma^{(2)}} g^{(2)} && \sigma^{(2)} = \sqrt{\frac{1}{3}[(g^{(0)})^2 + (g^{(1)})^2 + (g^{(2)})^2] + \epsilon} \\
&\ldots && \ldots \\
&\text{t=t+1: } w^{(t+1)} \leftarrow w^{(t)} - \frac{\eta^{(t)}}{\sigma^{(t)}} g^{(t)} && \sigma^{(t)} = \sqrt{\frac{1}{t+1} \sum_{i=1}^{t} (g^{(i)})^2 + \epsilon}
\end{aligned}
$$

ϵ is a smoothing term that avoids division by zero (usually on the order of $1e-8$)

Adagrad

- Divide the learning rate of each parameter by the root mean square of its previous derivatives

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t$$

where $\eta^t = \frac{\eta}{\sqrt{t+1}}$ and $\sigma^{(t)} = \sqrt{\frac{1}{t+1}\sum_{i=1}^{t}(g^{(i)})^2 + \epsilon}$.

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}} g^t$$

Contradiction?

- Vanilla gradient descent:

$$w^{t+1} \leftarrow w^t - \eta^t g^t$$

Larger gradient, larger step

- Adagrad:

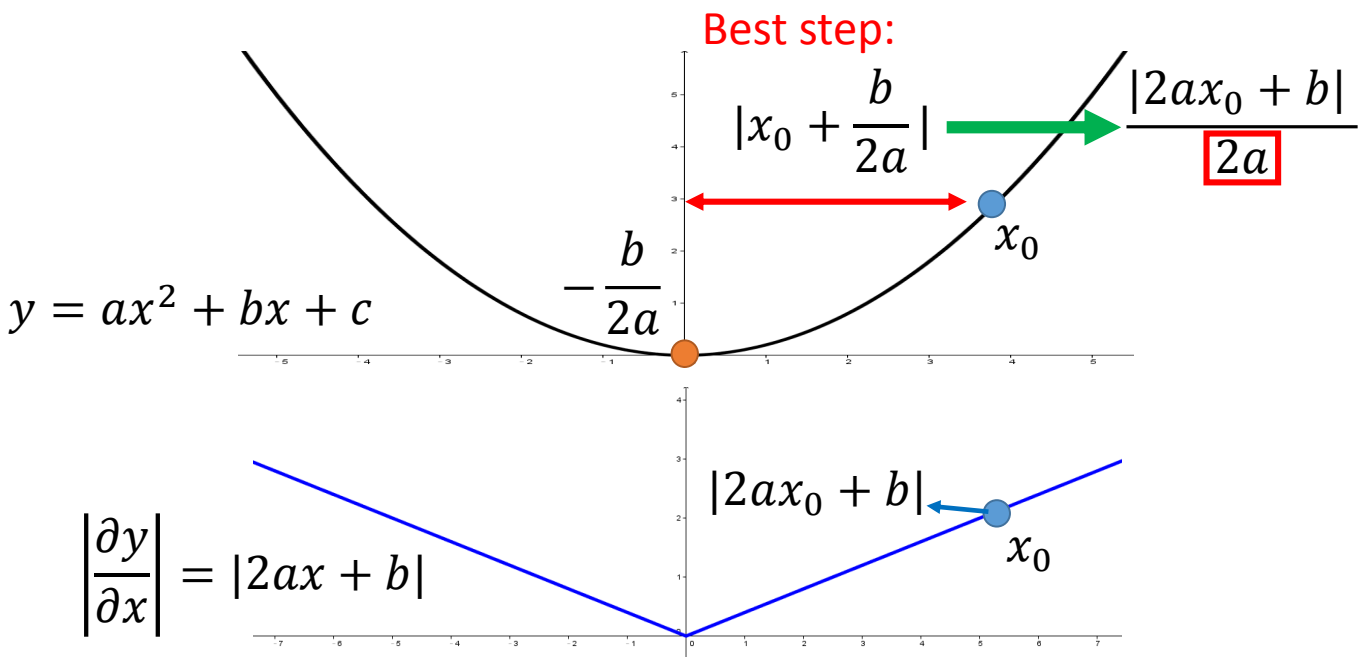
$$w^{t+1} \leftarrow w^t - \frac{\eta}{\underbrace{\sqrt{\sum_{i=0}^{t} (g^i)^2}}_{\text{larger gradient smaller step}}} \underbrace{g^t}_{\text{larger gradient larger step}}$$

Intuitive Reason

- Adagrad aims at capturing the relative changes compared to previous gradients (How surprise it is).

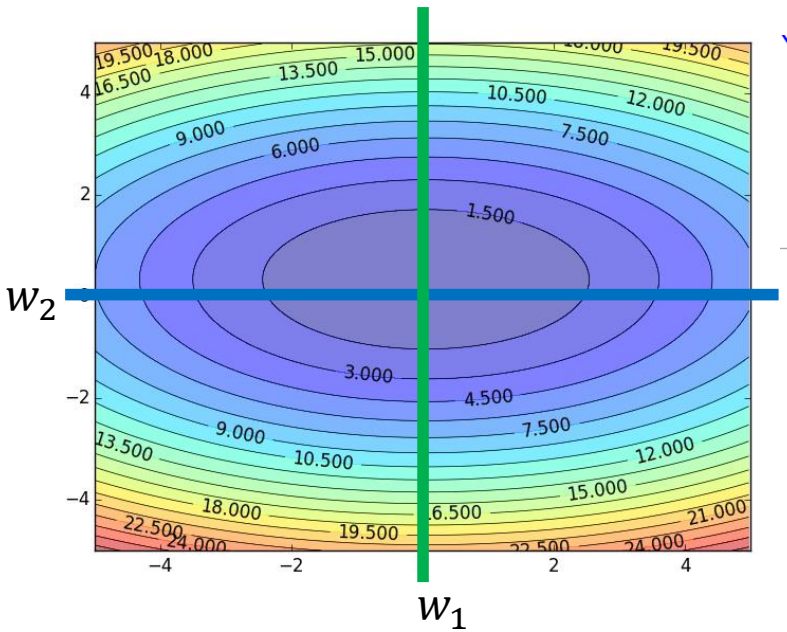
g^0	g^1	g^2	g^3	g^4	...
0.001	0.001	0.003	0.002	0.1	...
g^0	g^1	g^2	g^3	g^4	...
10.8	20.9	31.7	12.1	0.1	...

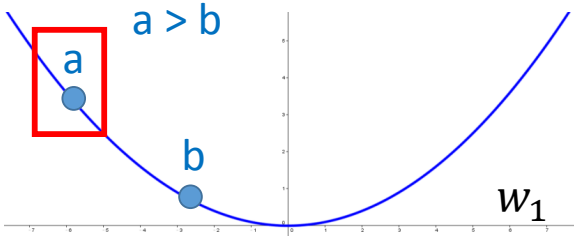
Second Derivative

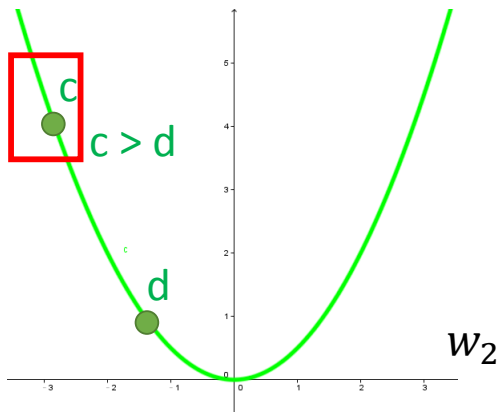


Note that $\frac{\partial^2 y}{\partial x^2} = 2a$. The best step is $\frac{|\text{first derivative}|}{\text{second derivative}}$

Comparison between different parameters

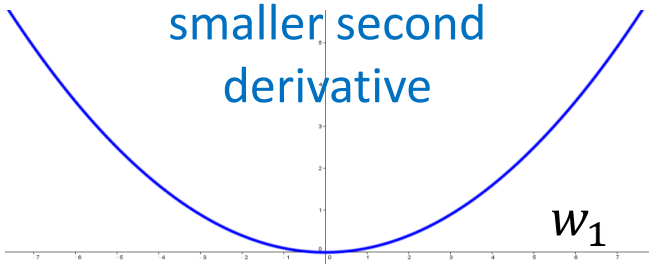


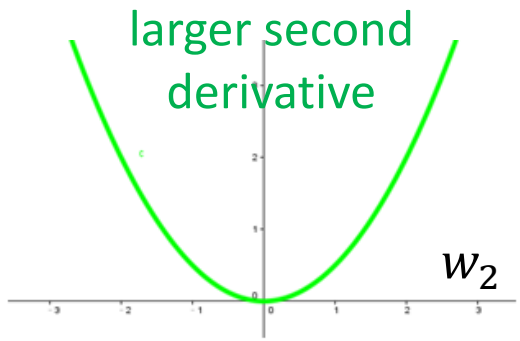




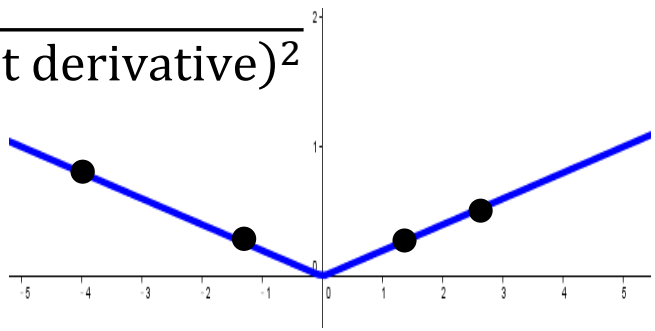
Second Derivative

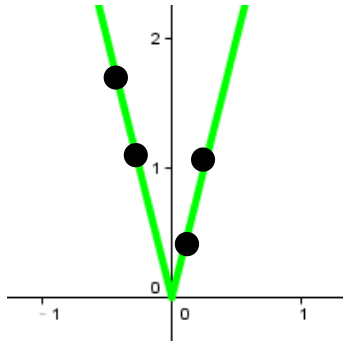
Use first derivative to estimate second derivative





$\sqrt{(\text{first derivative})^2}$





References

1. Visualization of Learning a Neural Network
2. Implementation of a 1-hidden layer Neural Network