

# Neural Networks

He He  
(adapted from David Rosenberg's slides)

CDS, NYU

April 28, 2020

# Contents

## 1 Overview

- Feature Learning
- Approximation Properties of Two-layer Neural Networks
- Multilayer Perceptron

## 2 Back-propagation

- Partial Derivatives and the Chain Rule
- Example Computation Graphs
- General Backpropagation

## 3 Advantages of Neural Networks

# Overview

- Learning non-linear models in a linear form:

$$f(x) = w^T \phi(x). \quad (1)$$

- What are possible  $\phi$ 's we have seen?
  - Feature maps that define a kernel, e.g., polynomials of  $x$
  - Feature templates, e.g.,  $x_1 = 1$  AND  $x_2 = 1$
  - Basis functions, e.g., (shallow) decision trees

# Decompose the problem

- Example:

**Task** Predict popularity of restaurants.

**Raw features** #dishes, price, wine option, zip code, #seats, size

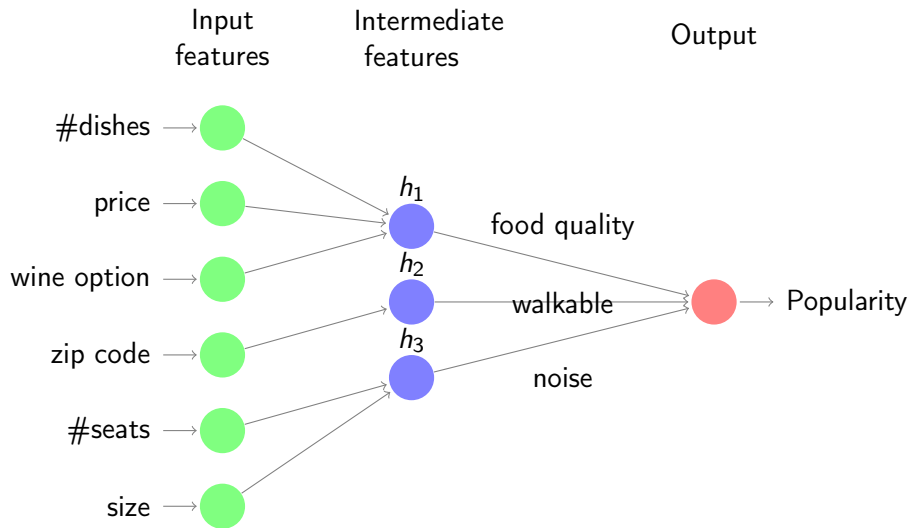
- Decompose into subproblems:

- $h_1$ ([#dishes, price, wine option]) = food quality
- $h_2$ ([zip code]) = walkable
- $h_3$ ([#seats, size]) = nosie

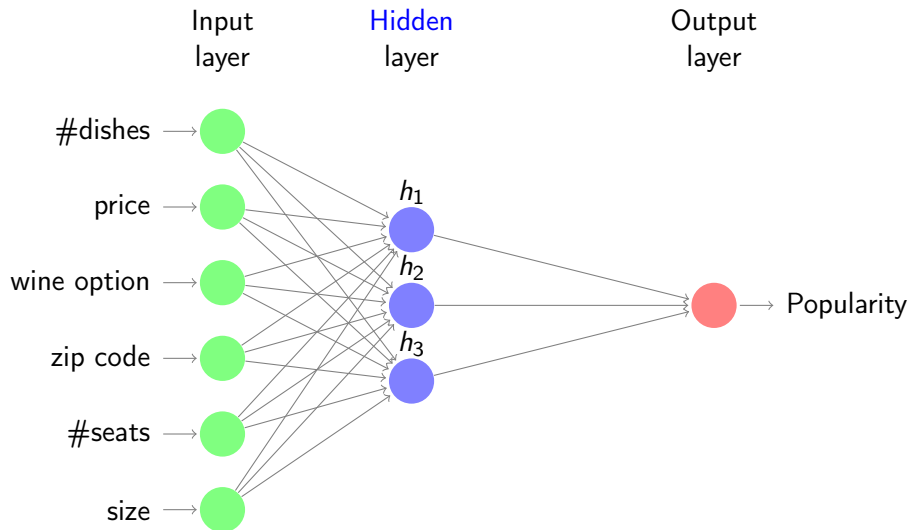
- Final *linear* predictor uses **intermediate features** computed by  $h_i$ 's:

$$w_1 \cdot \text{food quality} + w_2 \cdot \text{walkable} + w_3 \cdot \text{nosie}$$

# Predefined subproblems



## Learned intermediate features



**Key idea:** automatically learn the intermediate features.

**Feature engineering** Manually specify  $\phi(x)$  based on domain knowledge and learn the weights:

$$f(x) = \mathbf{w}^T \phi(x). \quad (2)$$

**Feature learning** Automatically learn both the features ( $K$  hidden units) and the weights:

$$h(x) = [\mathbf{h}_1(x), \dots, \mathbf{h}_K(x)], \quad (3)$$

$$f(x) = \mathbf{w}^T h(x) \quad (4)$$



# Activation function

- How should we parametrize  $h_i$ 's? Can it be linear?

$$h_i(x) = \sigma(v_i^T x). \quad (5)$$

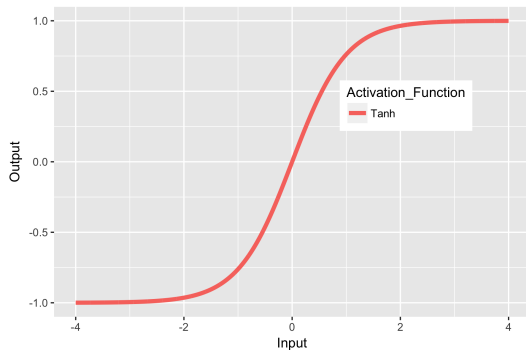
- $\sigma$  is the **nonlinear activation function**.
- What might be some activation functions we want to use?
  - sign function? **Non-differentiable**.
  - **Differentiable** approximations: sigmoid functions.
    - E.g., logistic function, hyperbolic tangent function.
- Two-layer neural network (one **hidden layer** and one **output layer**) with  $K$  hidden units:

$$f(x) = \sum_{k=1}^K w_k h_k(x) = \sum_{k=1}^K w_k \sigma(v_k^T x) \quad (6)$$

# Activation Functions

- The **hyperbolic tangent** is a common activation function:

$$\sigma(x) = \tanh(x).$$

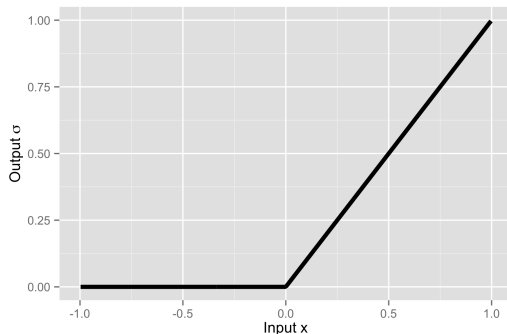


# Activation Functions

- More recently, the **rectified linear (ReLU)** function has been very popular:

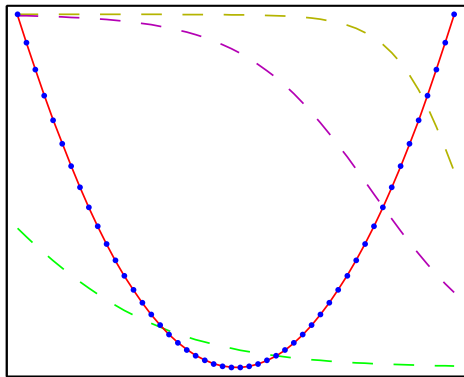
$$\sigma(x) = \max(0, x).$$

- Much **faster** to calculate, and to calculate its derivatives.
- Also often seems to work better.



## Approximation Ability: $f(x) = x^2$

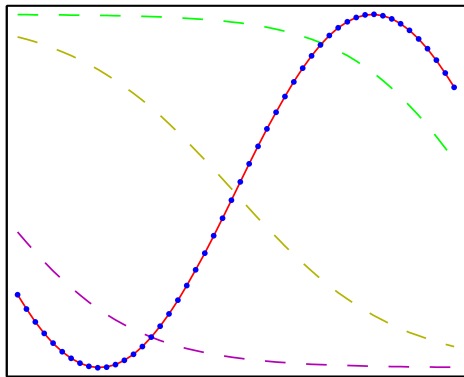
- 3 hidden units; tanh activation functions
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



From Bishop's [Pattern Recognition and Machine Learning](#), Fig 5.3

## Approximation Ability: $f(x) = \sin(x)$

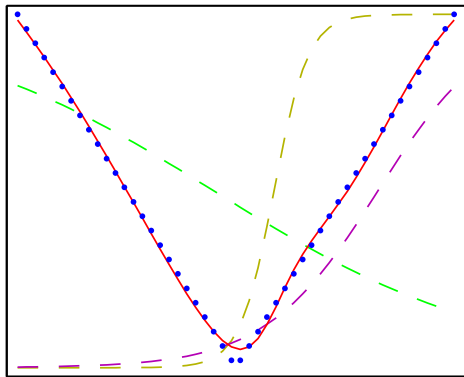
- 3 hidden units; logistic activation function
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



From Bishop's [Pattern Recognition and Machine Learning](#), Fig 5.3

## Approximation Ability: $f(x) = |x|$

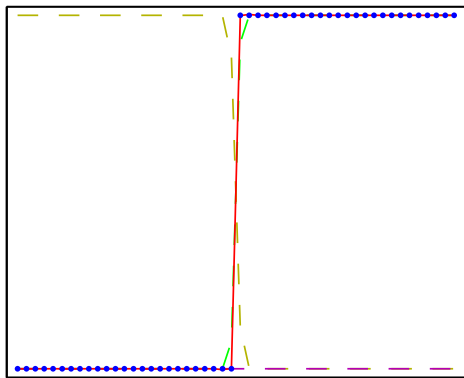
- 3 hidden units; logistic activation functions
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



From Bishop's [Pattern Recognition and Machine Learning](#), Fig 5.3

## Approximation Ability: $f(x) = 1(x > 0)$

- 3 hidden units; logistic activation function
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



From Bishop's [Pattern Recognition and Machine Learning](#), Fig 5.3

# Universal approximation theorems

How much expressive power do we gain from the nonlinearity?

## Theorem (Universal approximation theorem)

A neural network with *one possibly huge hidden layer*  $\hat{F}(x)$  can approximate *any* continuous function  $F(x)$  on a closed and bounded subset of  $\mathbb{R}^d$  under mild assumptions on the activation function, i.e.  $\forall \epsilon > 0$ , there exists an integer  $N$  s.t.

$$\hat{F}(x) = \sum_{i=1}^N w_i \sigma(v_i^T x + b_i) \quad (7)$$

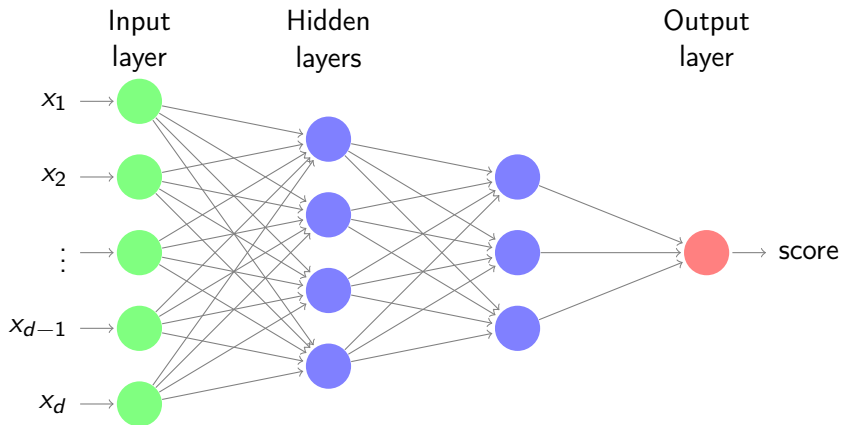
satisfies  $|\hat{F}(x) - F(x)| < \epsilon$ .

- The theorem doesn't tell us how to learn these parameters though!



# Multilayer perceptron / Feed-forward neural networks

- Wider: more hidden units.
- Deeper: more hidden layers.



# Multilayer Perceptron: Standard Recipe

- **Input space:**  $\mathcal{X} = \mathbb{R}^d$       **Action space**  $\mathcal{A} = \mathbb{R}^k$  (for  $k$ -class classification).
- Let  $\sigma: \mathbb{R} \rightarrow \mathbb{R}$  be an activation function (e.g. tanh or ReLU).
- Let's consider an MLP of  $L$  hidden layers, each having  $m$  hidden units.
- First hidden layer is given by

$$h^{(1)}(x) = \sigma \left( W^{(1)}x + b^{(1)} \right),$$

for parameters  $W^{(1)} \in \mathbb{R}^{m \times d}$  and  $b \in \mathbb{R}^m$ , and where  $\sigma(\cdot)$  is applied to each entry of its argument.

# Multilayer Perceptron: Standard Recipe

- Each subsequent hidden layer takes the **output**  $o \in \mathbb{R}^m$  of **previous layer** and produces

$$h^{(j)}(o^{(j-1)}) = \sigma\left(W^{(j)} o^{(j-1)} + b^{(j)}\right), \text{ for } j = 2, \dots, L$$

where  $W^{(j)} \in \mathbb{R}^{m \times m}$ ,  $b^{(j)} \in \mathbb{R}^m$ .

- Last layer is an **affine** mapping (no activation function):

$$a(o^{(L)}) = W^{(L+1)} o^{(L)} + b^{(L+1)},$$

where  $W^{(L+1)} \in \mathbb{R}^{k \times m}$  and  $b^{(L+1)} \in \mathbb{R}^k$ .

- The full neural network function is given by the **composition** of layers:

$$f(x) = \left(a \circ h^{(L)} \circ \dots \circ h^{(1)}\right)(x) \tag{8}$$

- Last layer typically gives us a score. How to do classification?

# Multinomial Logistic Regression

- From each  $x$ , we compute a linear score function for each class:

$$x \mapsto (\langle w_1, x \rangle, \dots, \langle w_k, x \rangle) \in \mathbb{R}^k$$

- We need to map this  $\mathbb{R}^k$  vector into a probability vector  $\theta$ .
- The **softmax function** maps scores  $s = (s_1, \dots, s_k) \in \mathbb{R}^k$  to a categorical distribution:

$$(s_1, \dots, s_k) \mapsto \theta = \mathbf{Softmax}(s_1, \dots, s_k) = \left( \frac{\exp(s_1)}{\sum_{i=1}^k \exp(s_i)}, \dots, \frac{\exp(s_k)}{\sum_{i=1}^k \exp(s_i)} \right)$$

# Nonlinear Generalization of Multinomial Logistic Regression

- From each  $x$ , we compute a non-linear score function for each class:

$$x \mapsto (f_1(x), \dots, f_k(x)) \in \mathbb{R}^k$$

where  $f_i$ 's are outputs of the last hidden layer of a neural network.

- Learning: Maximize the log-likelihood of training data

$$\arg \max_{f_1, \dots, f_k} \sum_{i=1}^n \log \left[ \text{Softmax}(f_1(x), \dots, f_k(x))_{y_i} \right].$$

- Unfortunately, this objective function is not concave or convex.
- But we can still use gradient methods to find a good local optimum.

# Neural network as a feature extractor

- OverFeat is a neural network for object classification, localization, and detection.
  - Trained on the huge ImageNet dataset
  - Lots of computing resources used for training the network.
- All those hidden layers of the network are very valuable **features**.
  - Paper: “**CNN Features off-the-shelf: an Astounding Baseline for Recognition**”
  - Showed that using features from OverFeat makes it easy to achieve state-of-the-art performance on new vision tasks.

We've seen

- Key idea: automatically discover useful features from raw data—**feature/representation learning**.
- Building blocks:
  - Input layer** no learnable parameters
  - Hidden layer(s)** perceptron + **nonlinear** activation function
  - Output layer** affine (+ transformation)
- A single hidden layer is sufficient to approximate any function.
- In practice, often have multiple hidden layers.

Next, how to learn the parameters.

# Back-propagation



# A brief history of artificial neural networks

## early 1940s–late 1960s

- Initial idea from neuroscience: create a computational model of neural networks.
- Development: perceptron [Rosenblatt, 1958], networks with many layers.
- Optimization: automatic differentiation [Linnainmaa, 1970].

## late 1960s–late 1980s

- Computers didn't have enough processing power [Minsky and Papert, 1969].
- Back-propagation invented [Werbos, 1975] (but still hard to train).
- AI research focused on expert systems and symbolic systems.

## late 1980s–early 2000s

- SVMs and linear models dominated ML.
- Continual developments in ANN: Schmidhuber, Hinton, Yann etc.

## Example: MLP Regression

- **Input space:**  $\mathcal{X} = \mathbb{R}$
- **Action Space / Output space:**  $\mathcal{A} = \mathcal{Y} = \mathbb{R}$
- **Hypothesis space:** MLPs with a single 3-node hidden layer:

$$f(x) = w_0 + w_1 h_1(x) + w_2 h_2(x) + w_3 h_3(x),$$

where

$$h_i(x) = \sigma(v_i x + b_i) \text{ for } i = 1, 2, 3,$$

for some fixed activation function  $\sigma: \mathbb{R} \rightarrow \mathbb{R}$ .

- What are the parameters we need to fit?

$$b_1, b_2, b_3, v_1, v_2, v_3, w_0, w_1, w_2, w_3 \in \mathbb{R}$$

# How to choose the best hypothesis?

- As usual, choose our prediction function using empirical risk minimization.
- Our hypothesis space is parameterized by

$$\theta = (b_1, b_2, b_3, v_1, v_2, v_3, w_0, w_1, w_2, w_3) \in \Theta = \mathbb{R}^{10}$$

- For a training set  $(x_1, y_1), \dots, (x_n, y_n)$ , find

$$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^{10}} \frac{1}{n} \sum_{i=1}^n (f(x_i; \theta) - y_i)^2.$$

- Gradient descent:
  - Is it differentiable w.r.t.  $\theta$ ?  $f(x) = w_0 + \sum_{i=1}^3 w_i \tanh(v_i x + b_i)$ .
  - Is it convex in  $\theta$ ? Might converge to a local minimum.

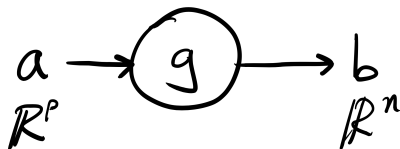
# Gradient descent for (large) neural networks

- Mathematically, it's just **partial derivatives**, which you can compute by hand using the **chain rule**.
  - In practice, this could be **time-consuming** and **error-prone**.
- How do we compute gradients in a systematic and efficient way?
  - **Back-propagation** (a special case of automatic differentiation).
  - Not limited to neural networks.
- Visualize with **computation graphs**.
  - Avoid long equations.
  - Structure of the computation (**modularity** and **dependency**), which allows for modern computation frameworks such as Tensorflow/Pytorch/MXNet/etc.

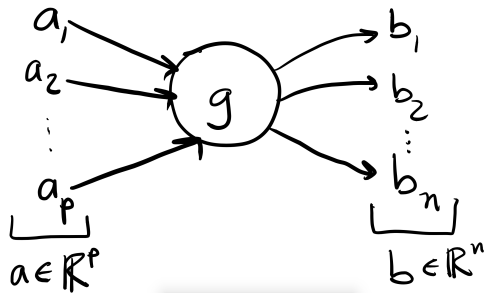
# Function as a graph

- Function as a **node** that takes in a set of **inputs** and produces a set of **outputs**.
- Example:  $g : \mathbb{R}^p \rightarrow \mathbb{R}^n$ .

- Typical computation graph:

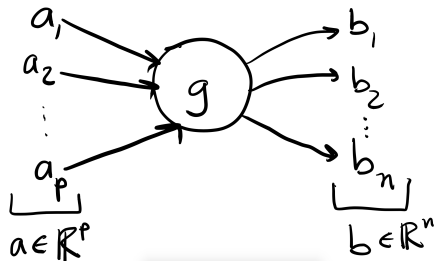


- Broken out into components:



# Partial Derivatives of an affine function

- Define the affine function  $g(x) = Mx + c$ , for  $M \in \mathbb{R}^{n \times p}$  and  $c \in \mathbb{R}$ .



- Let  $b = g(a) = Ma + c$ . What is  $b_i$ ?
- $b_i$  depends on the  $i$ th row of  $M$ :

$$b_i = \sum_{k=1}^p M_{ik} a_k + c_i.$$

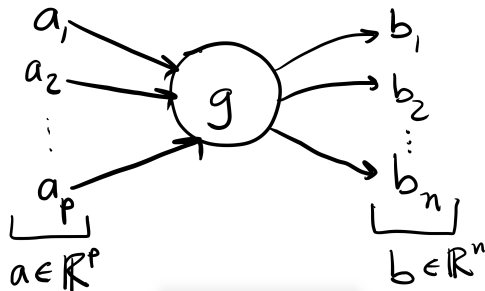
- If  $a_j \leftarrow a_j + \delta$ , what is  $b_i$ ?

$$b_i \leftarrow b_i + M_{ij} \delta.$$

Partial derivative/gradient measures **sensitivity**: If we perturb an input a little bit, how much does an output change?

# Partial Derivatives in general

- Consider a function  $g : \mathbb{R}^p \rightarrow \mathbb{R}^n$ .



- Partial derivative  $\frac{\partial b_i}{\partial a_j}$  is the instantaneous rate of change of  $b_i$  as we change  $a_j$ .
- If we change  $a_j$  slightly to

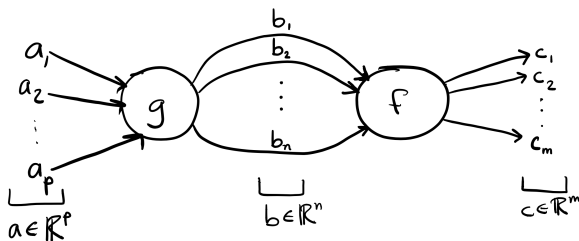
$$a_j + \delta,$$

- Then (for small  $\delta$ ),  $b_i$  changes to approximately

$$b_i + \frac{\partial b_i}{\partial a_j} \delta.$$

# Compose multiple functions

- Compose two functions  $g : \mathbb{R}^p \rightarrow \mathbb{R}^n$  and  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ .
- $b = g(a)$ ,  $c = f(b)$ .



- How does change in  $a_j$  affect  $c_i$ ?
- Visualize **chain rule**:
  - **Sum** changes induced on all paths from  $a_j$  to  $c_i$ .
  - Changes on one path is the **product** of changes on each edge along the path.

$$\frac{\partial c_i}{\partial a_j} = \sum_{k=1}^n \frac{\partial c_i}{\partial b_k} \frac{\partial b_k}{\partial a_j}.$$



## Example: Linear least squares

- Hypothesis space  $\{f(x) = w^T x + b \mid w \in \mathbb{R}^d, b \in \mathbb{R}\}$ .
- Data set  $(x_1, y_1), \dots, (x_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$ .
- Define

$$\ell_i(w, b) = [(w^T x_i + b) - y_i]^2.$$

- In SGD, in each round we'd choose a random index  $i \in 1, \dots, n$  and take a gradient step

$$\begin{aligned} w_j &\leftarrow w_j - \eta \frac{\partial \ell_i(w, b)}{\partial w_j}, \text{ for } j = 1, \dots, d \\ b &\leftarrow b - \eta \frac{\partial \ell_i(w, b)}{\partial b}, \end{aligned}$$

for some step size  $\eta > 0$ .

- Let's see how to calculate these partial derivatives on a computation graph.

# Computation Graph and Intermediate Variables

- For a generic training point  $(x, y)$ , denote the loss by

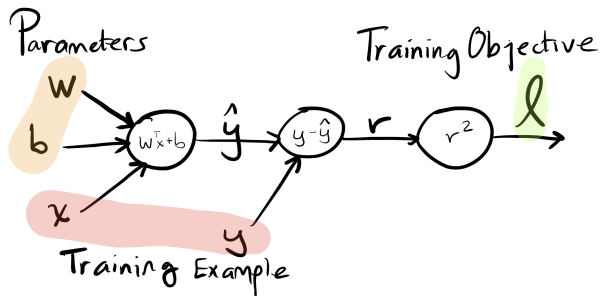
$$\ell(w, b) = [(w^T x + b) - y]^2.$$

- Let's break this down into some intermediate computations:

$$\text{(prediction)} \quad \hat{y} = \sum_{j=1}^d w_j x_j + b$$

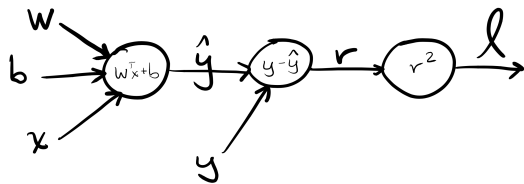
$$\text{(residual)} \quad r = y - \hat{y}$$

$$\text{(loss)} \quad \ell = r^2$$



# Partial Derivatives on Computation Graph

- We'll work our way from graph output  $\ell$  back to the parameters  $w$  and  $b$ :



$$\frac{\partial \ell}{\partial r} = 2r$$

$$\frac{\partial \ell}{\partial \hat{y}} = \frac{\partial \ell}{\partial r} \frac{\partial r}{\partial \hat{y}} = (2r)(-1) = -2r$$

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r$$

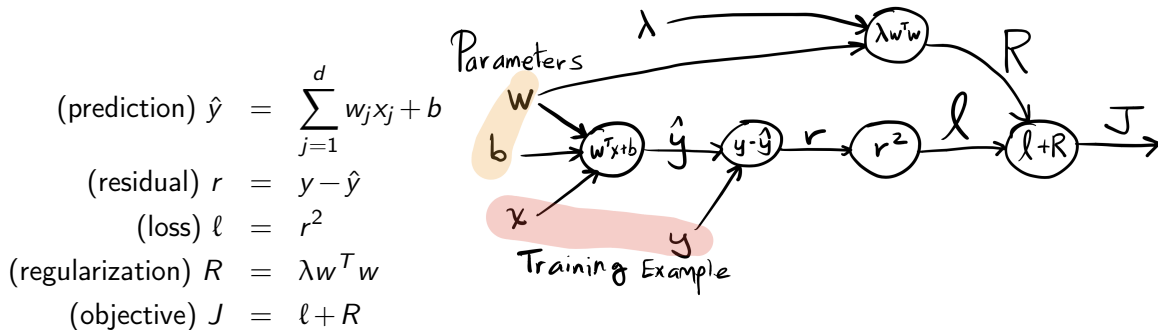
$$\frac{\partial \ell}{\partial w_j} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_j} = (-2r)x_j = -2rx_j$$

## Example: Ridge Regression

- For training point  $(x, y)$ , the  $\ell_2$ -regularized objective function is

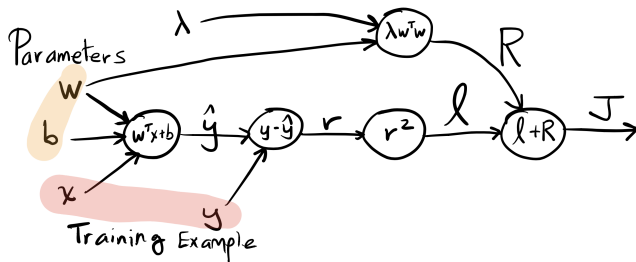
$$J(w, b) = [(w^T x + b) - y]^2 + \lambda w^T w.$$

- Let's break this down into some intermediate computations:



# Partial Derivatives on Computation Graph

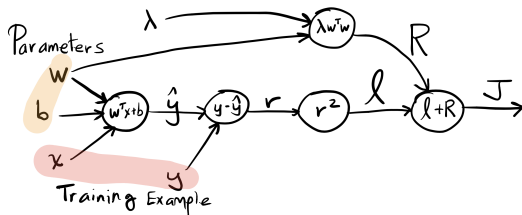
- We'll work our way from graph output  $\ell$  back to the parameters  $w$  and  $b$ :



$$\begin{aligned} \frac{\partial J}{\partial \ell} &= \frac{\partial J}{\partial R} = 1 \\ \frac{\partial J}{\partial \hat{y}} &= \frac{\partial J}{\partial \ell} \frac{\partial \ell}{\partial r} \frac{\partial r}{\partial \hat{y}} = (1)(2r)(-1) = -2r \\ \frac{\partial J}{\partial b} &= \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r \\ \frac{\partial J}{\partial w_j} &= \text{Exercise} \end{aligned}$$

# Backpropagation overview

- **Learning:** run gradient descent to find the parameters that minimize our objective  $J$ .
- Backpropagation: compute gradient w.r.t. each (trainable) parameter  $\frac{\partial J}{\partial \theta_i}$ .



**Forward pass** Compute intermediate function values, i.e. output of each node

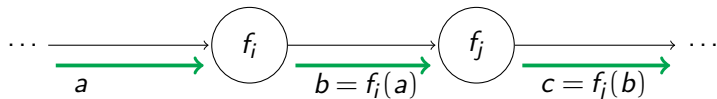
**Backward pass** Compute the partial derivative of  $J$  w.r.t. all intermediate variables and the model parameters

How to save computation?

- Path sharing: each node needs to **cache the intermediate results**.
- Think dynamic programming.

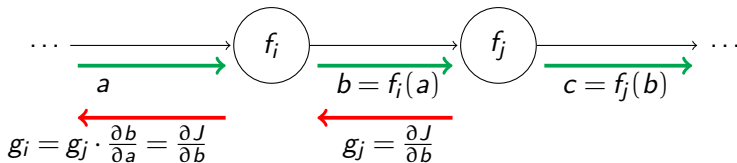
## Forward pass

- Order nodes by **topological sort** (every node appears before its children)
- For each node, compute the output given the input (output of its parents).
- Forward at intermediate node  $f_i$  and  $f_j$ :



## Backward pass

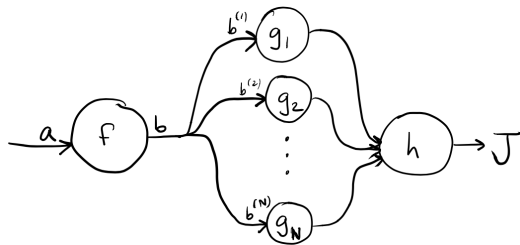
- Order nodes in **reverse topological order** (every node appear after its children)
- For each node, compute the partial derivative of its output w.r.t. its input, multiplied by the partial derivative from its children (chain rule).
- Backward at intermediate node  $f_i$ :





# Multiple children

- First sum partial derivatives from all children, then multiply.



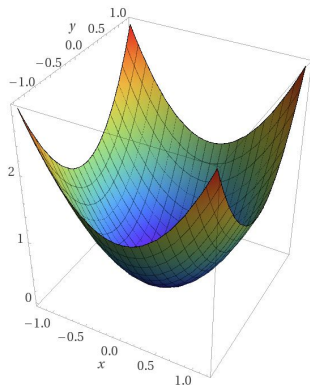
- Backprop for node  $f$ :
- Input:**  $\frac{\partial J}{\partial b^{(1)}}, \dots, \frac{\partial J}{\partial b^{(N)}}$   
(Partials w.r.t. inputs to all children)
- Output:**

$$\frac{\partial J}{\partial b} = \sum_{k=1}^N \frac{\partial J}{\partial b^{(k)}}$$
$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial b} \frac{\partial b}{\partial a}$$

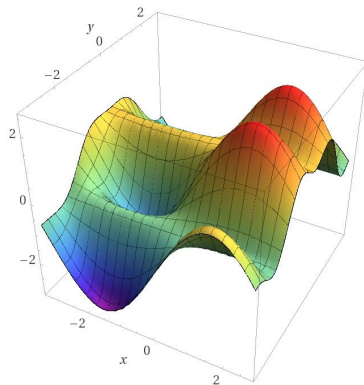
# Backpropagation in practice

- Inputs and outputs of nodes are generally **vectorized** (efficient to compute on GPUs).
- Computation graphs can be composed from a set of **basic operation nodes**, e.g., addition/multiplication, dot product, logistic function etc.
- Programming paradigms:
  - Symbolic** Specify all computation before data—efficient, e.g., Tensorflow.
  - Imperative** Specify the computation step by step—flexible/easier to write, e.g., Pytorch.
  - Hybrid** Can use either paradigm for computation subgraphs, e.g., MXNet.

# Non-convex optimization



Computed by Wolfram|Alpha



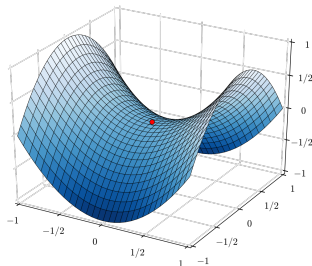
Computed by Wolfram|Alpha

- Left: convex loss function. Right: non-convex loss function.

# Non-convex optimization: challenges

Optimization of neural networks is generally hard.

- Converge to a bad local minimum.
  - Try different initialization and rerun.
- Saddle point.
  - Doesn't often happen with SGD.
  - Second partial derivative test.
- “Flat” region: low gradient magnitude
  - Use ReLU instead of sigmoid as activation functions.
- High curvature: high gradient magnitude
  - Gradient clipping.
  - Adaptive step sizes.



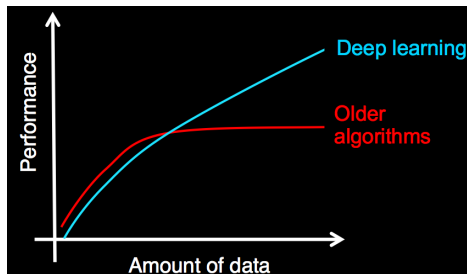
---

Reference: Chris De Sa's slides (CS6787 Lecture 7).

- Backpropagation is an algorithm to compute gradient (partial derivatives + chain rule) efficiently.
- It is used in gradient descent optimization with neural networks.
- Key idea: function composition and dynamic programming
- In practice, efficient software exists (backpropagation, neural network building blocks, optimization algorithms etc.).

## Advantages of Neural Networks

# Neural Networks Benefit from Big Data



- Empirical observation: performance of deep neural networks has not plateaued with increasing amount of data.
- Higher data throughput compared to other nonlinear methods.
- System for ML: how to run (training/inference) large neural networks efficiently.

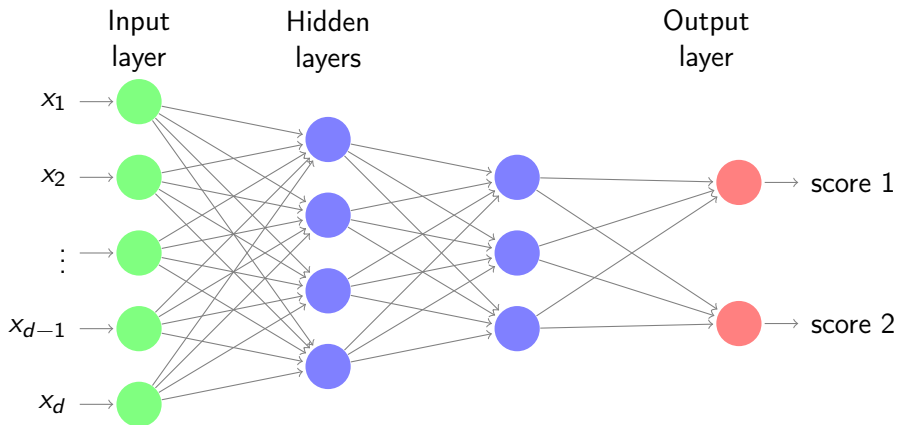
From Andrew Ng's CS229 Deep Learning slides (<http://cs229.stanford.edu/materials/CS229-DeepLearning.pdf>)

- Easy to incorporate inductive bias for different tasks.
- Examples:
  - Translation invariance in image recognition—convolution.
  - Long-range dependence in language—recurrence.
  - Alignment between input and (structured) output—attention.
- Many building blocks than can be composed together.



# Representation sharing

- “Classifiers” are task-specific but representation/features can be shared.



# Multitask Learning and transfer learning

## Multitask learning:

- Learn related tasks together, e.g.,
  - Object classification: cat or dog?
  - Object localization: location of the objects?
- Basic features (e.g., edges, texture) can be shared by tasks.
  - Different output layers for each task; the rest is shared.
  - Objective function combines losses from both predictions, e.g. by averaging.

## Transfer learning:

- Self-supervised **pre-training** to learn generic features.
  - General idea: denoising, i.e. perturbed input  $\rightarrow$  original input.
- On downstream tasks: **fine-tune** pre-trained models (reuse representation).

# Summary

- Powerful use of features: representation learning
- Hard to train: non-convex optimization
  - Easier now with released code and libraries!
- Fast and scalable: with the right system support.
- Gap exists between theory and practice.