

Feature learning, neural networks and backpropagation

CDS, NYU

April 18, 2022

Today's lecture

- Neural networks: huge empirical success but poor theoretical understanding
- Key idea: representation learning
- Optimization: backpropagation + SGD

- Many problems are non-linear
- We can express certain non-linear models in a linear form:

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

- Note that this model is not linear in the inputs x — we represent the inputs differently, and the new representation is amenable to linear modeling
- For example, we can use a feature map that defines a kernel, e.g., polynomials in x

Decomposing the problem

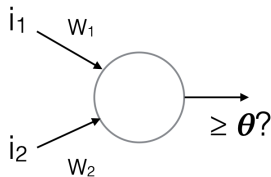
- Example: predicting how popular a restaurant is
Raw features #dishes, price, wine option, zip code, #seats, size
- Decomposing the problem into subproblems:
 - $h_1([\text{\#dishes}, \text{price}, \text{wine option}]) = \text{food quality}$
 - $h_2([\text{zip code}]) = \text{walkable}$
 - $h_3([\text{\#seats}, \text{size}]) = \text{noisy}$
- Each intermediate models solves one of the subproblems
- A final *linear* predictor uses the **intermediate features** computed by the h_i 's:

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

Perceptrons as logical gates

- Suppose that our input features indicate light at a two points in space (0 = no light; 1 = light)
- How can we build a perceptron that detects when there is light in both locations?

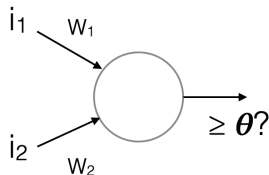
$$w_1 = 1, w_2 = 1, \theta = 2$$



i_1	i_2	$w_1 i_1 + w_2 i_2$
0	0	0
0	1	1
1	0	1
1	1	2

Limitations of a perceptrons as logical gates

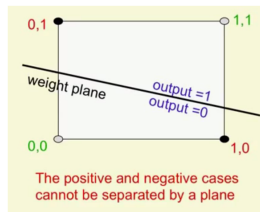
- Can we build a perceptron that fires when the two pixels have the same value ($i_1 = i_2$)?



Positive: (1, 1) (0, 0)

$$\begin{aligned} w_1 + w_2 &\geq \theta, & 0 &\geq \theta \\ w_1 < \theta, & & w_2 < \theta \end{aligned}$$

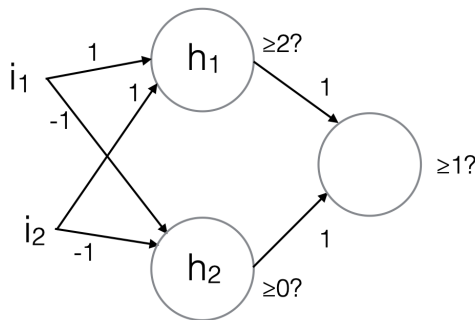
Negative: (1, 0) (0, 1)



If θ is negative, the sum of two numbers that are both less than θ cannot be greater than θ

Multilayer perceptron

- Fire when the two pixels have the same value ($i_1 = i_2$)

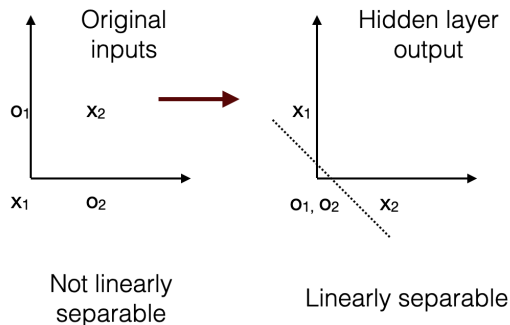


	Hidden layer input				Hidden layer output		
	i_1	i_2	h_1	h_2	h_1	h_2	o
x_1	0	0	0	0	0	1	1
o_1	0	1	1	-1	0	0	0
o_2	1	0	1	-1	0	0	0
x_2	1	1	2	-2	1	0	1

(for x_1 and x_2 the correct output is 1;
for o_1 and o_2 the correct output is 0)

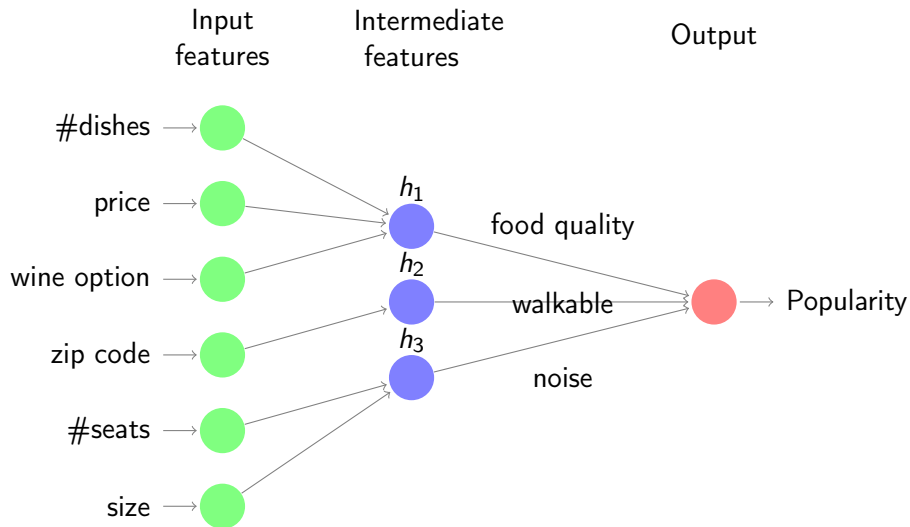
Multilayer perceptron

- Recode the input: the hidden layer representations are now linearly separable

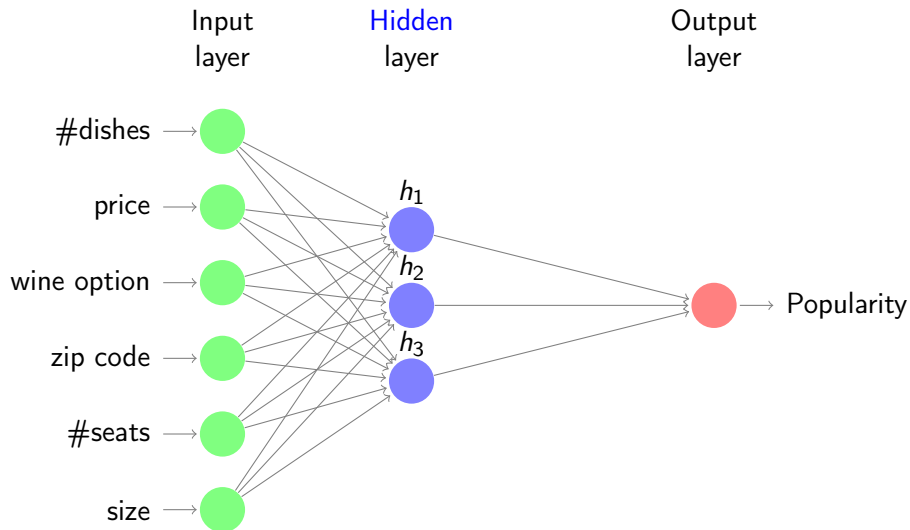


			Hidden layer input		Hidden layer output		
	i_1	i_2	h_1	h_2	h_1	h_2	o
x_1	0	0	0	0	0	1	1
o_1	0	1	1	-1	0	0	0
o_2	1	0	1	-1	0	0	0
x_2	1	1	2	-2	1	0	1

Decomposing the problem into predefined subproblems



Learned intermediate features



Key idea: learn the intermediate features.

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

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

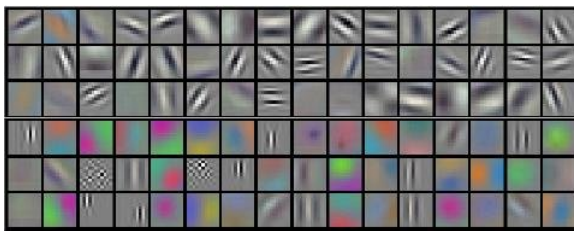
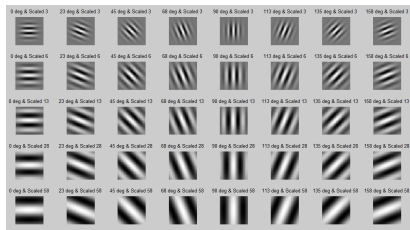
Feature learning Learn both the features (K hidden units) and the weights:

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

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

Feature learning example

- A filter convolves over the image and looks for the highest pattern match.
- Traditionally, people use Gabor filters or other image feature extractors, e.g. SIFT, SURF, etc, and an SVM on top for image classification.
- Neural networks take in images and can learn the filters that are the most useful for solving the tasks. Likely more efficient than hand engineered features.



Inspiration: The brain

- Our brain has about 100 billion (10^{11}) neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons, with non-linear computations.

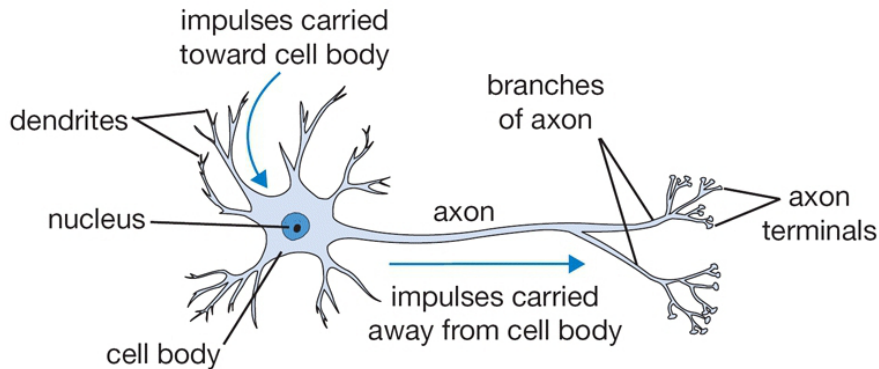
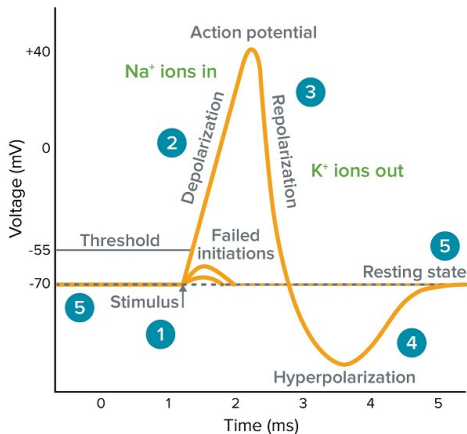


Figure: The basic computational unit of the brain: Neuron

Inspiration: The brain

- Neurons receive input signals and accumulate voltage. After some threshold they will fire spiking responses.



Activation function

- We can model a simpler computation by using “activation function”.
- It applies a non-linearity on the inputs and “fires” after some threshold.

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

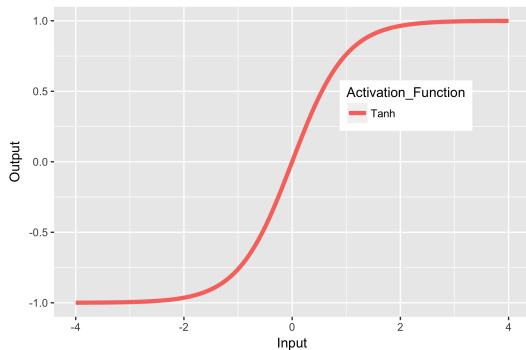
- Some possible activation functions:
 - sign function (as in classic perceptron)? **Non-differentiable**.
 - *Differentiable* approximations: sigmoid functions.
 - E.g., logistic function, hyperbolic tangent function.
- Two-layer neural network (one **hidden layer** and one **output layer**) with K 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:

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

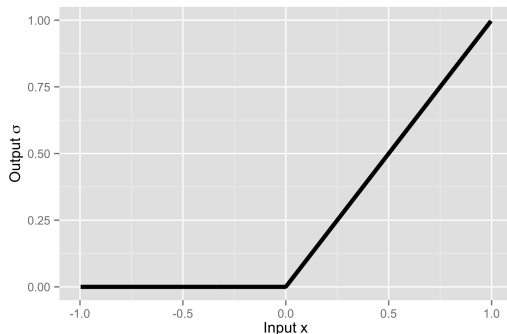


Activation Functions

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

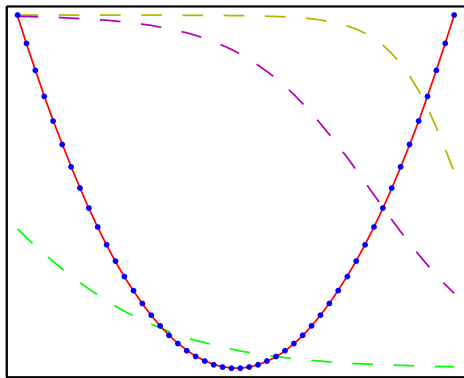
$$\sigma(x) = \max(0, x).$$

- Faster to calculate this function and its derivatives
- Often more effective in practice



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

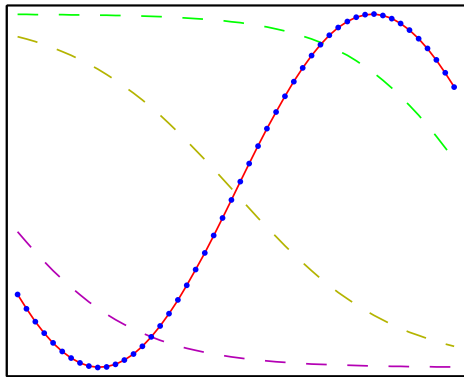
- 3 hidden units; tanh activation functions
- Blue dots are training points; dashed lines are hidden unit outputs; final output in red.



From Bishop's *Pattern Recognition and Machine Learning*, Fig 5.3

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

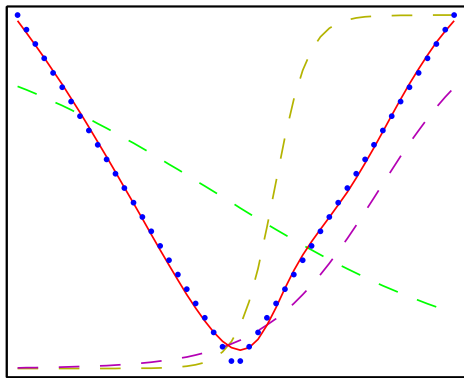
- 3 hidden units; logistic activation function
- Blue dots are training points; dashed lines are hidden unit outputs; final output in red.



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

Universal approximation theorem

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.

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

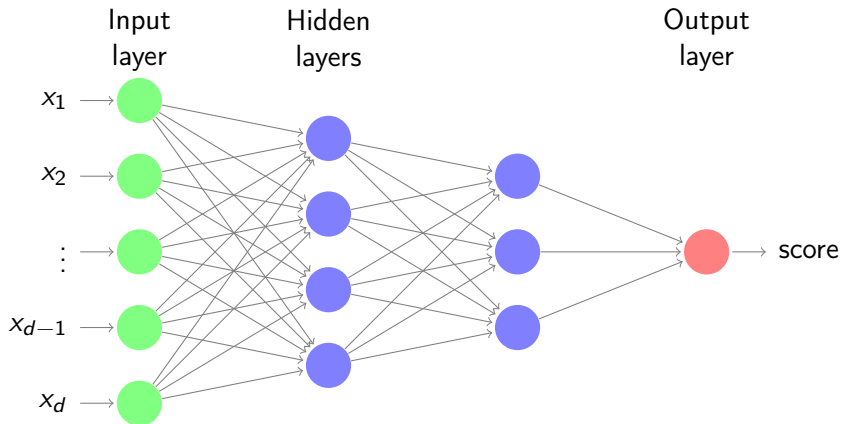
satisfies $|\hat{F}(x) - F(x)| < \epsilon$.

Universal approximation theorem

- For the theorem to work, the number of hidden units needs to be exponential in d
- The theorem doesn't tell us how to find the parameters of this network
- It doesn't explain why practical neural networks work, or tell us how to build them

Deep neural networks

- Wider: more hidden units (as in the approximation theorem).
- Deeper: more hidden layers.



Multilayer Perceptron (MLP): formal definition

- **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 (MLP): formal definition

- 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:

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

- Typically, the last layer gives us a score. How do we perform classification?

What did we do in 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 θ .
- 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 the outputs of the last hidden layer of a neural network.

- Learning: Maximize the log-likelihood of training data

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

Interim discussion

- With the right representations, we can turn nonlinear problems into linear ones
- The goal of representation learning is to automatically discover useful features from raw data
- Building blocks:
 - Input layer no learnable parameters
 - Hidden layer(s) affine + *nonlinear* activation function
 - Output layer affine (+ softmax)
- A single, potentially huge hidden layer is sufficient to approximate any function
- In practice, it is often helpful to have multiple hidden layers

Fitting the parameters of an MLP

- **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?

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

Finding the best hypothesis

- As usual, we 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 $(x_1, y_1), \dots, (x_n, y_n)$, our goal is to find

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

How do we learn these parameters?

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

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

- We can use gradient descent
- Is f differentiable w.r.t. θ ? $f(x) = w_0 + \sum_{i=1}^3 w_i \tanh(v_i x + b_i)$.
- Is the loss convex in θ ?
 - \tanh is not convex
 - Regardless of nonlinearity, the composition of convex functions is not necessarily convex
- We might converge to a local minimum.

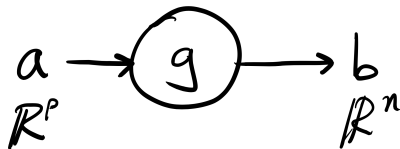
Gradient descent for (large) neural networks

- 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**
- Back-propagation computes gradients for neural networks (and other models) in a systematic and efficient way
- We can visualize the process using *computation graphs*, which expose the structure of the computation (**modularity** and **dependency**)

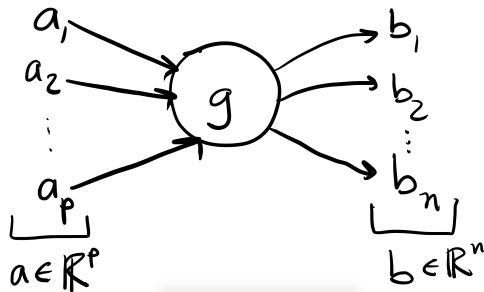
Functions as nodes in a graph

- We represent each component of the network 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:

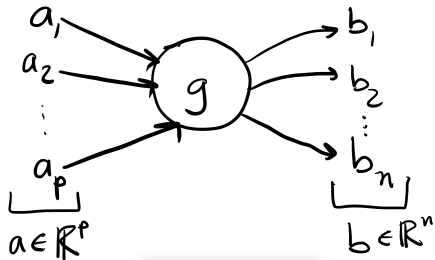


- Broken down by component:



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 :

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

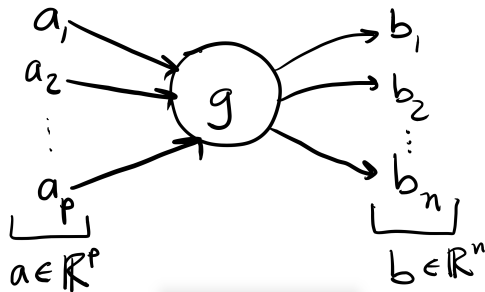
- If $a_j \leftarrow a_j + \delta$, what is b_i ?

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

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

Partial derivatives in general

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

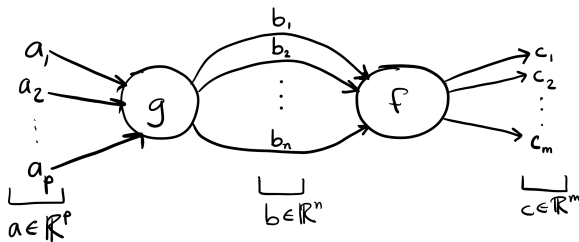


- Partial derivative $\frac{\partial b_i}{\partial a_j}$ is the rate of change of b_i as we change a_j
- If we change a_j slightly to
$$a_j + \delta,$$
- Then (for small δ), b_i changes to approximately

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

Composing multiple functions

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



- How does a small change in a_j affect c_i ?
- Visualizing the **chain rule**:
 - We **sum** changes induced on all paths from a_j to c_i .
 - The change contributed by each 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 choose a random training instance $i \in 1, \dots, n$ and take a gradient step

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

$$b \leftarrow b - \eta \frac{\partial \ell_i(w, b)}{\partial b},$$

for some step size $\eta > 0$.

- How do we calculate these partial derivatives on a computation graph?

Computation graph and intermediate variables

- For a training point (x, y) , the loss is

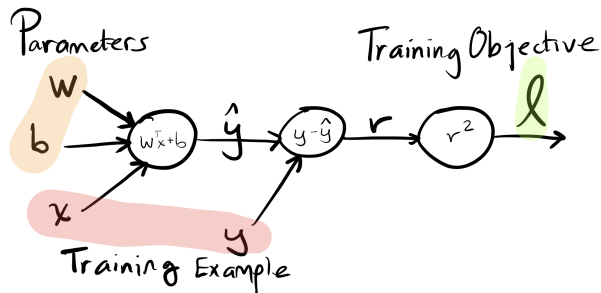
$$\ell(w, b) = [(w^T x + b) - y]^2.$$

- Let's break this down into intermediate computations:

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

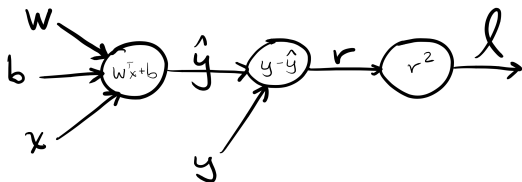
$$\text{(residual)} \quad r = y - \hat{y}$$

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



Partial derivatives on computation graph

- We'll work our way from the output ℓ back to the parameters w and b , reusing previous computations as much as possible:



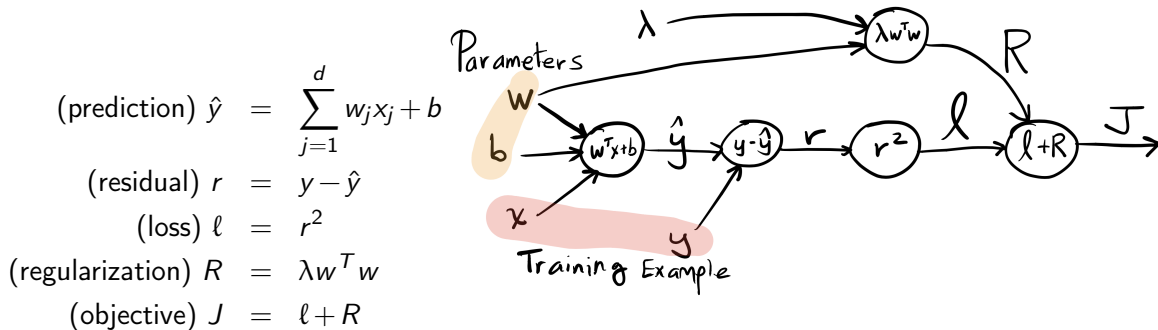
$$\begin{aligned}\frac{\partial \ell}{\partial r} &= 2r \\ \frac{\partial \ell}{\partial \hat{y}} &= \frac{\partial \ell}{\partial r} \frac{\partial r}{\partial \hat{y}} = (2r)(-1) = -2r \\ \frac{\partial \ell}{\partial b} &= \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r \\ \frac{\partial \ell}{\partial w_j} &= \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_j} = (-2r)x_j = -2rx_j\end{aligned}$$

Example: Ridge Regression

- For training point (x, y) , the ℓ_2 -regularized objective function is

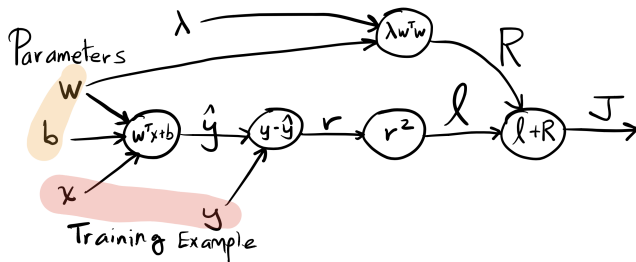
$$J(w, b) = [(w^T x + b) - y]^2 + \lambda w^T w.$$

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



Partial Derivatives on Computation Graph

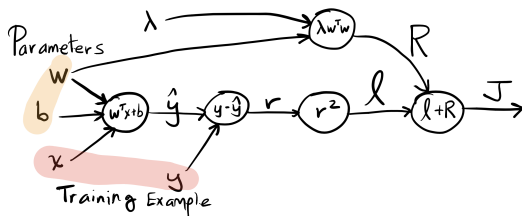
- We'll work our way from graph output ℓ back to the parameters w and b :



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

Backpropagation: Overview

- **Learning:** run gradient descent to find the parameters that minimize our objective J .
- Backpropagation: we compute the 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

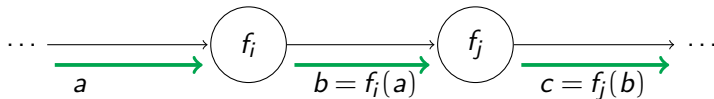
Backward pass Compute the partial derivative of J w.r.t. all intermediate variables and the model parameters

How do we minimize computation?

- Path sharing: each node *caches intermediate results*: we don't need to compute them over and over again
- An example of dynamic programming

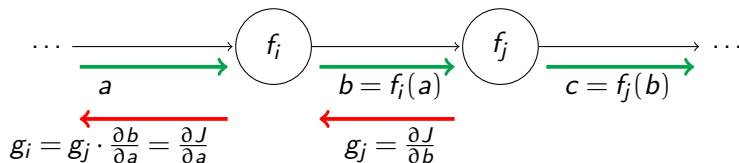
Forward pass

- 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 :



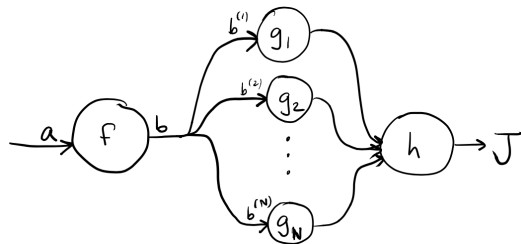
Backward pass

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

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

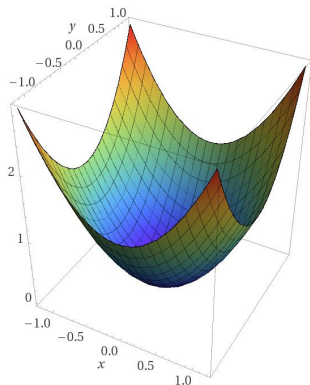
Why backward?

- We can write the gradient in terms of chain rule.

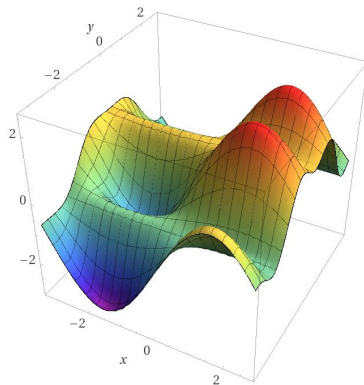
a

(9)

Non-convex optimization



Computed by Wolfram|Alpha

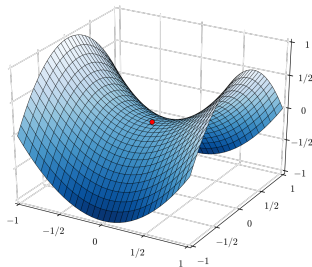


Computed by Wolfram|Alpha

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

Non-convex optimization: challenges

- What if we converge to a bad local minimum?
 - Rerun with a different initialization
- Hit a saddle point
 - Doesn't often happen with SGD
 - Second partial derivative test
- Flat region: low gradient magnitude
 - Possible solution: use ReLU instead of sigmoid
- High curvature: large gradient magnitude
 - Possible solutions: Gradient clipping, adaptive step sizes



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

Learning rate

- One of the most important hyperparameter.
- Start with a higher learning rate then decay towards zero.
- Classic theory: convergence guarantee for stochastic gradient descent. Otherwise the update step has a noise term dominated by the noise of data sample.
- Other explanation: Loss surface, avoidance of local minima, avoidance of memorization of noisy samples
- Learning rate decay (staircase 10x, cosine, etc.)

More in-depth topics

- Neural networks have been a popular machine learning model in the past decade, due to its learning capability.
- Today: MLP + SGD
- Adding structures: weight-sharing, convolution, recurrent, residual connection, attention
- Learning conditioning: normalization, optimizers
- System: GPU/TPU, parallel computing, gradient aggregation
- Some topics covered in DS-GA 1008 Deep Learning
- Most of the advances built on top of backpropagation

- Backprop is used to train the overwhelming majority of neural nets today.
- Despite its practical success, backprop is believed to be neurally implausible.
- No evidence for biological signals analogous to error derivatives.
- Forward & backward weights are tied in backprop.
- Backprop requires synchronous update (1 forward followed by 1 backward).
- Biologically plausible alternatives we know about learn much more slowly on computers.

- Backpropagation is an algorithm for computing the gradient (partial derivatives + chain rule) efficiently
- It is used in gradient descent optimization for neural networks
- Key idea: function composition and the chain rule
- In practice, we can use existing software packages, e.g. PyTorch (backpropagation, neural network building blocks, optimization algorithms etc.)