Ch03. Multi-Layer Perceptrons (MLPs):
Fully-connected Neural Networks and Backpropagation

Hwanjo Yu

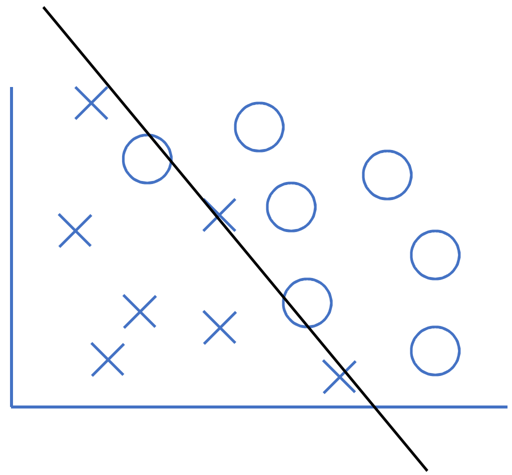
POSTECH

http://hwanjoyu.org

Outline

- Perceptron: A single layer neural network.
- Multilayer perceptron (MLP): A multilayer extension of perceptron.
    - Universal approximation
    - Error back-propagation (BP) algorithm

Linear Classification



- A linear discriminant function which has the form
$$f(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b,$$
- where $(\mathbf{w}, b) \in \mathbb{R}^D \times \mathbb{R}$ (weight vector, bias) are the parameters that control the function.
- Decision rule is given by $\mathrm{sgn}\big(f(\mathbf{x})\big)$,
$$\mathrm{sgn}\big(f(\mathbf{x})\big) = \begin{cases} 1 & \text{if } f(x) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Separating Hyperplane

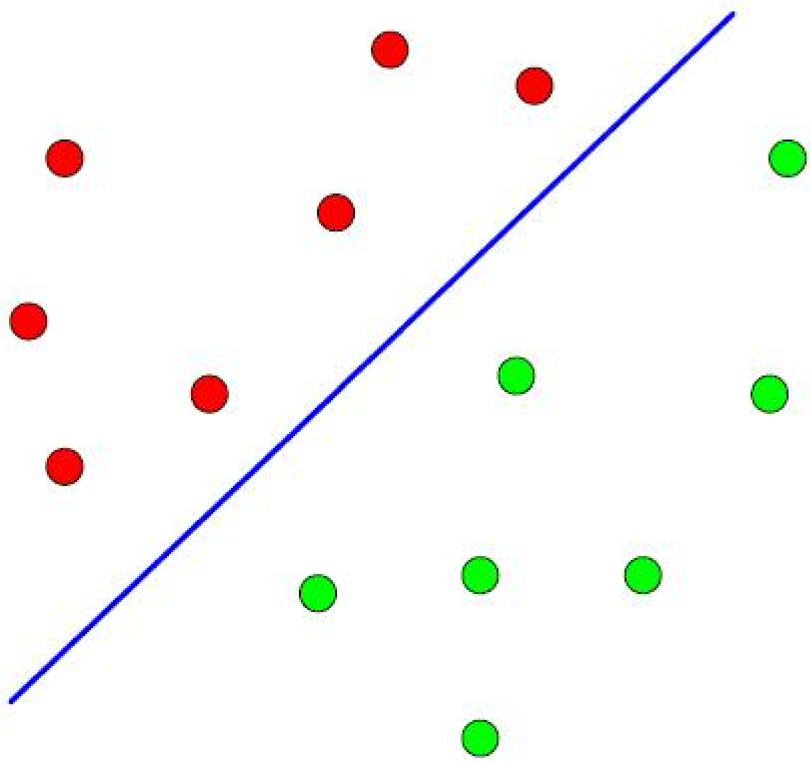
- A separating hyperplane is defined by
$$\mathbf{w}^{\mathrm{T}}\mathbf{x} + b = 0.$$
- The input space $\mathcal{X}$ is split into two parts by the hyperplane.
- The separating hyperplane is an affine subspace of dimension $D - 1$ which divides the space into two half spaces which corresponds to the inputs of the two distinct classes.

Perceptron

- Proposed by Rosenblatt in 1956
- The first iterative algorithm for learning linear classification
- A single-layer neural network with threshold activation function:
$$\mathbf{y} = \mathrm{sgn}(\mathbf{w}^{\mathrm{T}}\mathbf{x} + b)$$
- On-line and mistake-driven procedure: The weight vector is updated each time a training point is misclassified.
- Perceptron Convergence: The algorithm is guaranteed to converge when data are linearly separable.

Linearly Separable

- Two classes of patterns are "linearly separable" if they can be separated by a linear hyperplane.
- In other words, there exists a hyperplane which separates two classes.



Perceptron Criterion

- Suppose that target values $\{y_n\}$ take either 1 or -1:

$$y_n = \begin{cases} 1 & \text{if } \mathbf{x}_n \in \mathcal{C}_1 \\ -1 & \text{if } \mathbf{x}_n \in \mathcal{C}_2 \end{cases}$$

- What we want here is to find a $\mathbf{w}$ such that

$$\begin{cases} \mathbf{w}^{\mathrm{T}}\mathbf{x}_n > 0 & \text{if } \mathbf{x}_n \in \mathcal{C}_1 \\ \mathbf{w}^{\mathrm{T}}\mathbf{x}_n < 0 & \text{if } \mathbf{x}_n \in \mathcal{C}_2 \end{cases}$$

- which is identical to

$$\mathbf{w}^{\mathrm{T}}\mathbf{x}_n y_n > 0 \quad \forall \mathbf{x}_n.$$

- The perceptron criterion leads to the following objective function

$$\mathcal{J}(\mathbf{w}) = -\sum_{\mathbf{x}_n \in \mathcal{M}} \mathbf{w}^{\mathrm{T}}\mathbf{x}_n y_n$$

- where $\mathcal{M}$ is the set of vectors $\mathbf{x}_n$ which are misclassified by the current weight vector.
- The gradient of $\mathcal{J}(\mathbf{w})$ is

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = -\sum_{\mathbf{x}_n \in \mathcal{M}} \mathbf{x}_n y_n$$

Perceptron Learning: Algorithm Outline

1. Get a training sample
2. If classified correctly, do nothing.

   If classified incorrectly, update $\mathbf{w}$ by

$$\mathbf{w}_{(k+1)} = \mathbf{w}_{(k)} + \alpha \mathbf{x}_n y_n$$

3. Repeat steps 1 and 2 until convergence

Perceptron Convergence Theorem

- The perceptron classifier minimizes the error probability.
- One can easily see that the perceptron learning reduce the error

$$
\begin{aligned}
-\mathbf{w}_{(k+1)}^{\mathrm{T}}\mathbf{x}_n y_n &= -\mathbf{w}_{(k)}^{\mathrm{T}}\mathbf{x}_n y_n - \sum_{\mathbf{x}_n \in \mathcal{M}} \|\mathbf{x}_n y_n\|^2 \\
&\leq -\mathbf{w}_{(k)}^{\mathrm{T}}\mathbf{x}_n y_n
\end{aligned}
$$

- Theorem: *If classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are linearly separable, then the perceptron rule converges in a finite number of steps to a separating hyperplane.*
- In their book, "Perceptrons: An Introduction to Computational Geometry", **Minsky and Papert** (1969) showed that a perceptron can't solve the XOR problem.
- This contributed the first AI winter.

Multi-Layer Perceptron (MLP)

Motivation

Example: predicting car collision

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

True function: safe if cars sufficiently far

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

Examples:

x	y
[1,3]	+1
[3,1]	+1
[1,0.5]	-1

Decomposing the problem

Test if car 1 is far right of car 2:

$$h_1 = \mathbf{1}[x_1 - x_2 \geq 1]$$

Test if car 2 is far right of car 1:

$$h_2 = \mathbf{1}[x_2 - x_1 \geq 1]$$

Safe if at least one is true:

$$y = \text{sign}(h_1 + h_2)$$

x	h_1	h_2	y
[1,3]	0	1	+1
[3,1]	1	0	+1
[1,0.5]	0	0	-1

Learning strategy

Define: $\phi(x) = [1, x_1, x_2]$:

Intermediate hidden subproblems:

$$h_1 = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_1 = [-1, +1, -1]$$

$$h_2 = \mathbf{1}[\mathbf{v}_2 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_2 = [-1, -1, +1]$$

Final prediction:

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

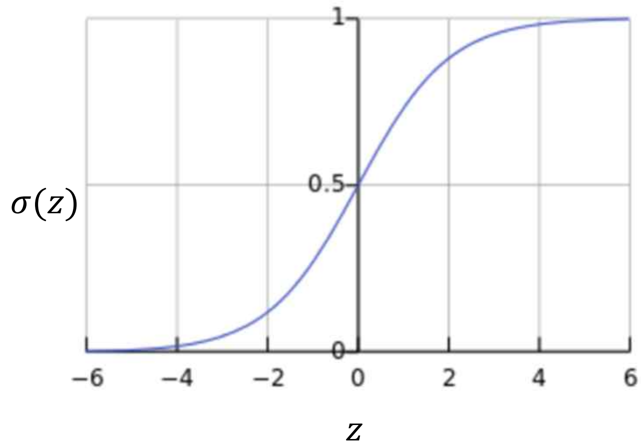
Key idea: joint learning

- Goal: learn both hidden subproblems $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2)$ and combination weights $\mathbf{w} = [w_1, w_2]$

Gradients

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

$$h_1 = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$$



Definition: logistic function (or sigmoid function)

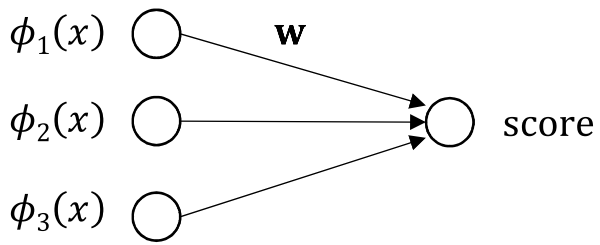
- The logistic function maps $(-\infty, \infty)$ to [0,1]:
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$
- Derivative of sigmoid:
$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Solution:

$$h_1 = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

Linear predictors

Linear predictor:

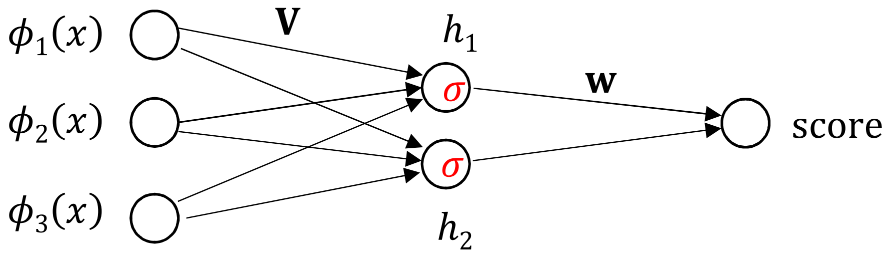


Output:

$$\text{score} = \mathbf{w} \cdot \phi(x)$$

Neural networks

Neural network:



Intermediate hidden units:

$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \qquad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot \mathbf{h}$$

Note: In neural network, σ is called activation function. Traditionally the sigmoid function was used, but recently the **rectified linear function σ(z) = max{z,0}** has gained popularity.

Neural networks

Think of intermediate hidden units as learned features of a linear predictor.

Key idea: feature learning

- Before: apply linear predictor on manually specified features

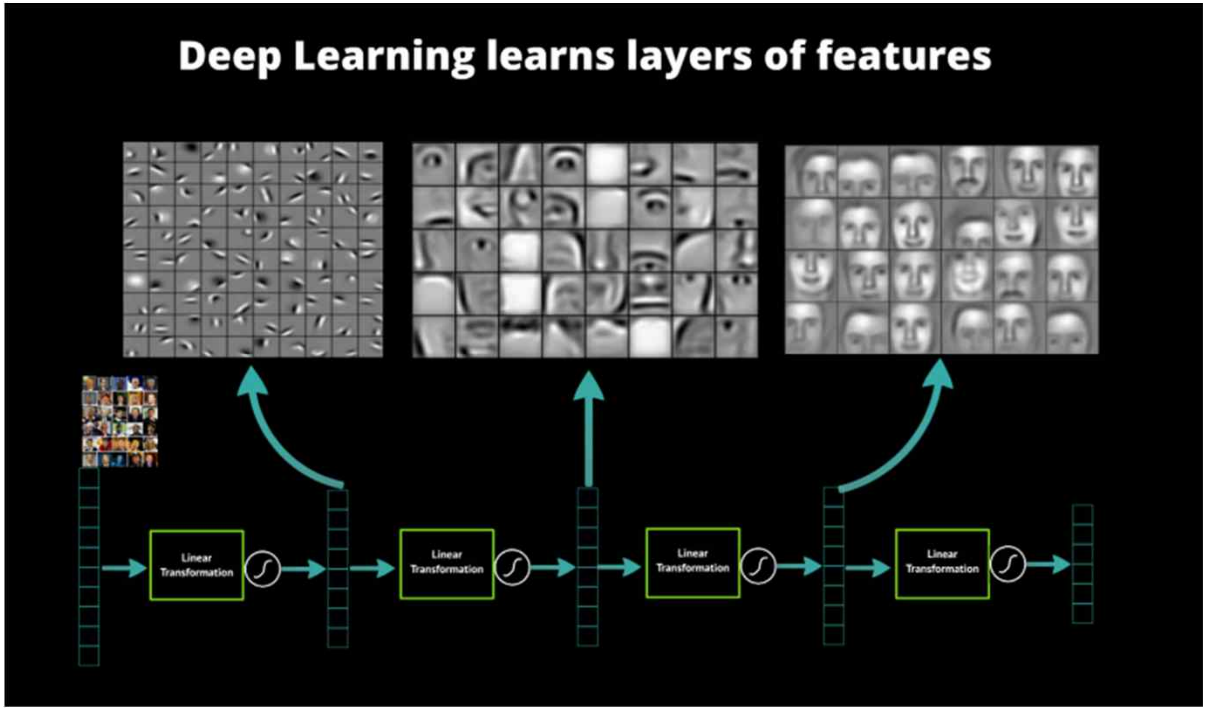
$$\phi(x)$$

- Now: apply linear predictor on automatically learned features

$$h(x) = [h_1(x), \ldots, h_k(x)]$$

Question: can the functions $h_j = \sigma(\mathbf{v}_j \cdot \phi(x))$ supply good features for a linear predictor?

Deep Learning



Back Propagation

Motivation: loss minimization

Optimization problem:

- $\min_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V},\mathbf{w})$
- $\text{TrainLoss}(\mathbf{V},\mathbf{w}) = \frac{1}{|D_{\text{train}}|}\sum_{(x,y)\in D_{\text{train}}} \text{Loss}(x,y,\mathbf{V},\mathbf{w})$
- $\text{Loss}(x,y,\mathbf{V},\mathbf{w}) = \left(y - f_{\mathbf{V},\mathbf{w}}(x)\right)^2$
- $f_{\mathbf{V},\mathbf{w}}(x) = \sum_{j=1}^{d} w_j \sigma(\mathbf{v}_j \cdot \phi(x))$

Goal: compute gradient

$$\nabla_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V},\mathbf{w})$$

=> Doable but tedious

Approach

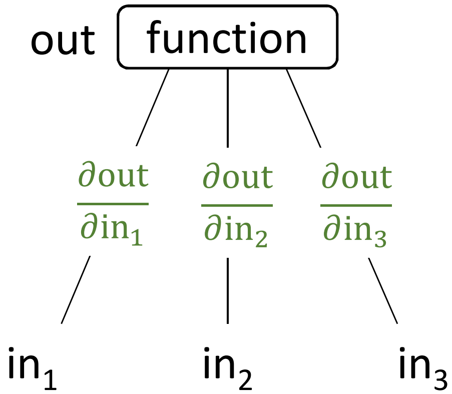
Mathematically: just grind through the chain rule

Next: visualize the computation using a computation graph

Advantage:

- Avoid long equations
- Reveal structure of computations (modularity, efficiency, dependencies)

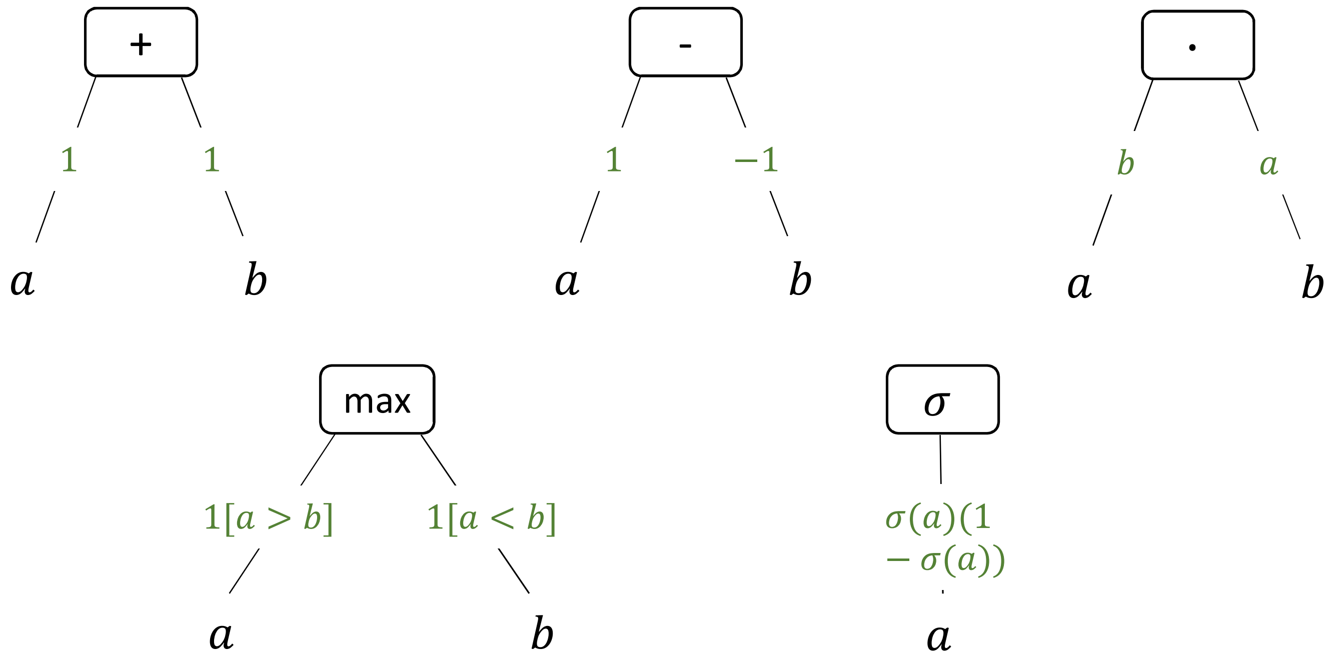
Function as boxes



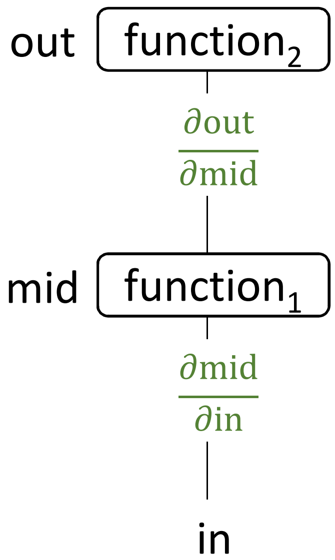
Partial derivatives (gradients): how much does the output change if an input changes?

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

Basic building blocks



Composing functions

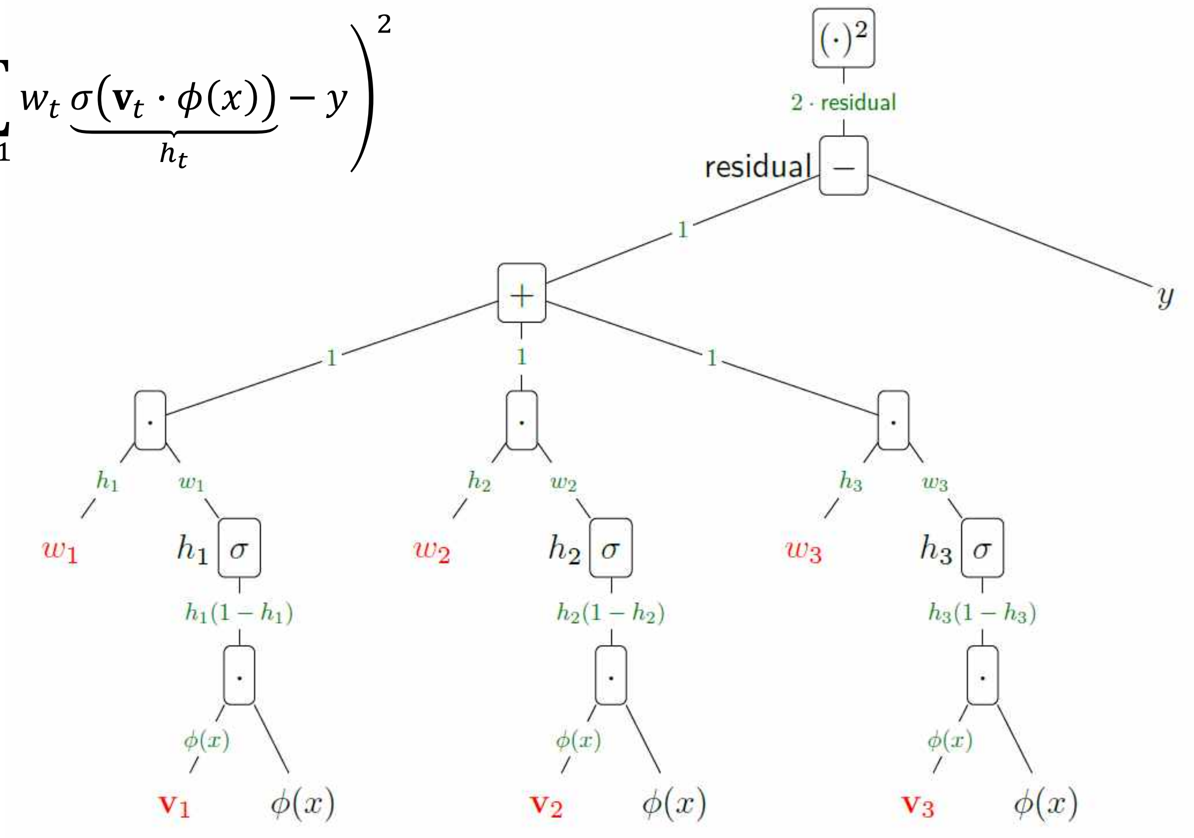


Chain rule:

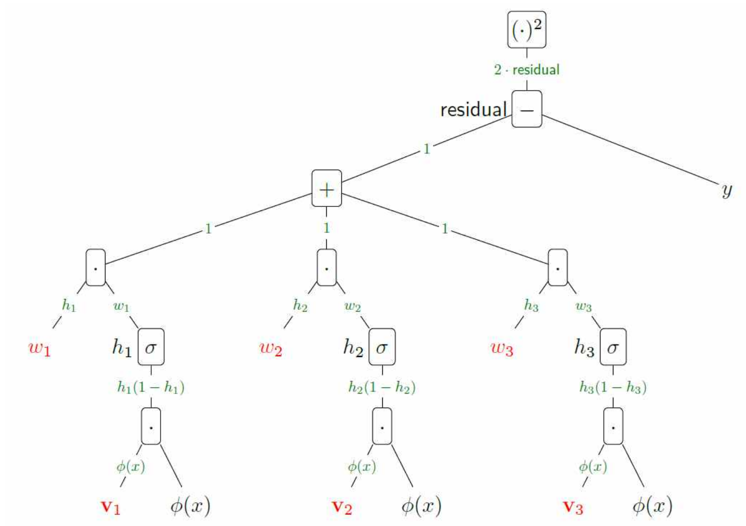
$$\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$$

Neural network

$$\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = \left(\sum_{t=1}^{d} w_t \underbrace{\sigma(\mathbf{v}_t \cdot \phi(x))}_{h_t} - y \right)^2$$



Backpropagation



out

$\frac{\partial out}{\partial f^{(3)}}$

$f^{(3)}$ $\quad g^{(3)} = \frac{\partial out}{\partial f^{(3)}}$

$\frac{\partial f^{(3)}}{\partial f^{(2)}}$

$f^{(2)}$ $\quad g^{(2)} = \frac{\partial f^{(3)}}{\partial f^{(2)}} g^{(3)}$

$\frac{\partial f^{(2)}}{\partial f^{(1)}}$

$f^{(1)}$ $\quad g^{(1)} = \frac{\partial f^{(2)}}{\partial f^{(1)}} g^{(2)}$

$\frac{\partial f^{(1)}}{\partial in}$

in $\quad \frac{\partial out}{\partial in} = \frac{\partial f^{(1)}}{\partial in} g^{(1)}$

Algorithm: backpropagation

- Forward pass: compute each f (from leaves to root)
- Backward pass: compute each g (from root to leaves)

Forward: $f^{(k)}$ is value for subexpression rooted at k.

⇑6 $out = \left(f^{(5)}\right)^2$

⇑5 $f^{(5)} = f^{(4)} - y$

⇑4 $f^{(4)} = \sum_{t=1}^{d} f_t^{(3)}$

⇑3 $f_1^{(3)} = w_1 f_1^{(2)}$

⇑2 $f_1^{(2)} = \sigma(f_1^{(1)}) = h_1$

⇑1 $f_1^{(1)} = \mathrm{v}_1 \phi(x)$

Backward: $g^{(k)} = \frac{\partial out}{\partial f^{(k)}}$ is how $f^{(k)}$ influences output.

⇓1 $g^{(5)} = \frac{\partial out}{\partial f^{(5)}} = 2f^{(5)}$

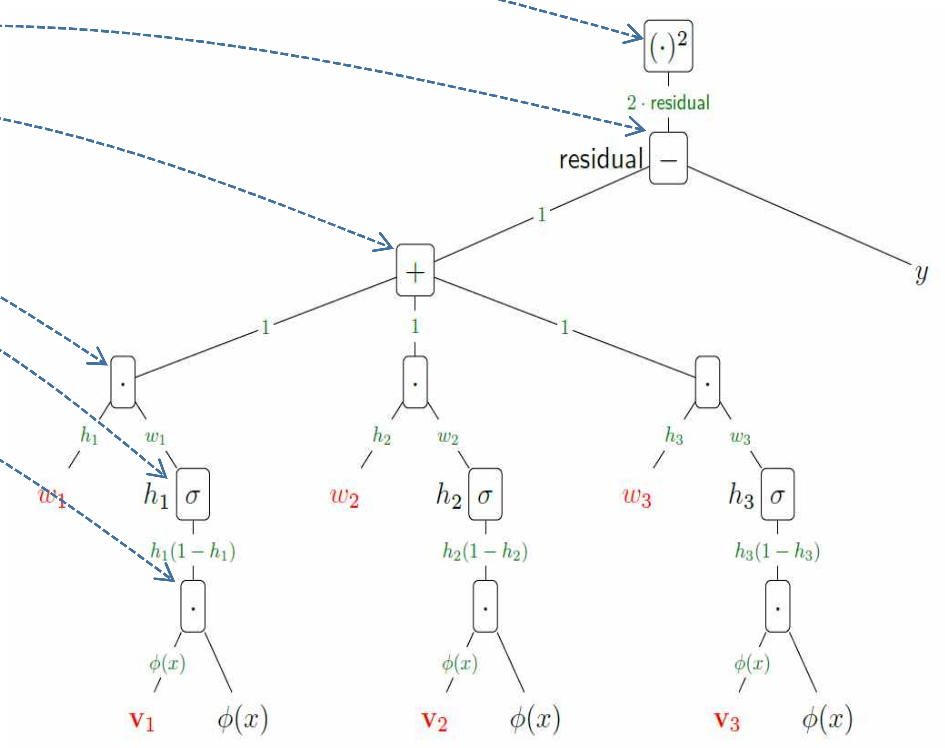
⇓2 $g^{(4)} = \frac{\partial out}{\partial f^{(4)}} = \frac{\partial f^{(5)}}{\partial f^{(4)}} \cdot g^{(5)} = 2f^{(5)}$

⇓3 $g_1^{(3)} = \frac{\partial out}{\partial f_1^{(3)}} = \frac{\partial f^{(4)}}{\partial f_1^{(3)}} \cdot g^{(4)} = 2f^{(5)}$

⇓4 $g_1^{(2)} = \frac{\partial out}{\partial f_1^{(2)}} = \frac{\partial f_1^{(3)}}{\partial f_1^{(2)}} \cdot g_1^{(3)} = w_1 2f^{(5)}$

⇓5 $g_1^{(1)} = \frac{\partial out}{\partial f_1^{(1)}} = \frac{\partial f_1^{(2)}}{\partial f_1^{(1)}} \cdot g_1^{(2)} = h_1(1-h_1)w_1 2f^{(5)}$

⇓6 $\frac{\partial out}{\partial \mathrm{v}_1} = \frac{\partial f_1^{(1)}}{\partial \mathrm{v}_1} \cdot g_1^{(1)} = \phi(x)h_1(1-h_1)w_1 2f^{(5)}$



Optimization with backpropagation

Algorithm: GD

- Initialize $\mathbf{w} = [0, \ldots, 0]$
- For $t = 1, \ldots, T$:
  $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

Algorithm: SGD

- Initialize $\mathbf{w} = [0, \ldots, 0]$
- For $t = 1, \ldots, T$:
  For each $(x, y) \in D_{\text{train}}$:
  $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(\mathbf{w})$

- $\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = \left(\sum_{t=1}^{d} w_t \underbrace{\sigma\left(\mathbf{v}_t \cdot \phi(x)\right)}_{h_t} - y\right)^2$

$= (\mathbf{w} \cdot \mathbf{h} - y)^2$

- $\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2 \cdot (f(x) - y) \cdot \mathbf{h}$
- $\nabla_{\mathbf{v}_1} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2 \cdot (f(x) - y) \cdot w_1 h_1 (1 - h_1) \cdot \phi(x)$
- $\nabla_{\mathbf{v}_2} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2 \cdot (f(x) - y) \cdot w_2 h_2 (1 - h_2) \cdot \phi(x)$
- …

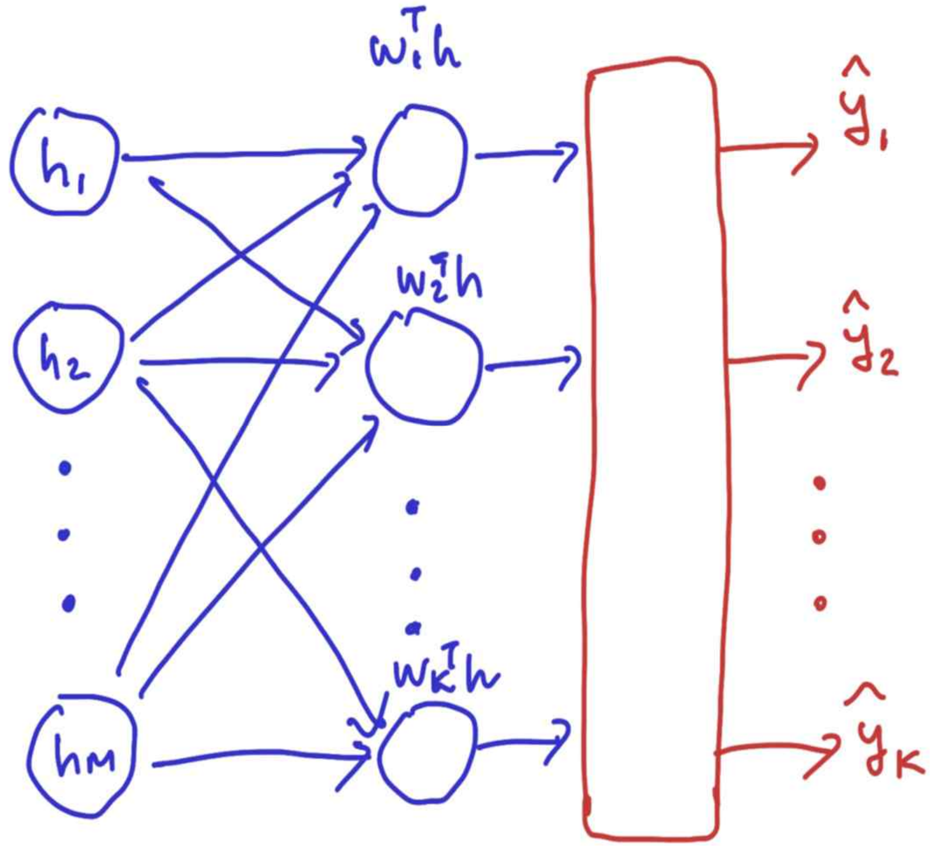
Error Functions

- Denote by $f(\mathbf{x};\theta)$ a neural network parameterized by θ, which takes $\mathbf{x}$ as input.
- Denote $\hat{y}_n$ by the output of the neural network when input $\mathbf{x}_n$ is given. The corresponding target is y_n.
- Squared error for regression:

$$\frac{1}{2N}\sum_{n=1}^{N}\|y_n - \hat{y}_n\|^2 .$$

- Cross entropy error for classification: See next slides

Softmax Layers



$$\hat{y}_k = \frac{\exp(\mathbf{w}_k^{\mathrm{T}}\mathbf{h})}{\sum_{j=1}^{K} \exp(\mathbf{w}_j^{\mathrm{T}}\mathbf{h})}$$

Cross Entropy Error

- In the case of logistic regression, we have
$$p \in \{y, 1-y\}, \qquad q \in \{\hat{y}, 1-\hat{y}\}.$$
- Then, the **cross entropy error** is calculated as
$$\mathcal{E} = \sum_{n=1}^{N} \left[-y_n \log \hat{y}_n - (1-y_n)\log(1-\hat{y}_n)\right].$$

- In the case of softmax regression, we have
$$p \in \{y_1, y_2, \dots y_K\}, \qquad q \in \{\hat{y}_1, \hat{y}_2, \dots, \hat{y}_K\}.$$
- Then, the **cross entropy error** is calculated as
$$\mathcal{E} = \sum_{n=1}^{N} \sum_{k=1}^{K} \left[-y_{k,n} \log \hat{y}_{k,n}\right].$$

Cross Entropy Error

$$\text{Model} \longrightarrow \hat{y} = \begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix} \qquad \begin{array}{c} \text{Target} \\ y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \end{array} \begin{array}{l} \\ \text{Tiger} \\ \text{Lion} \\ \text{Cat} \end{array}$$

$$\begin{aligned} CE &= -1 \log 0.7 - 0 \log 0.2 - 0 \log 0.1 \\ &= -\log 0.7 \\ &= 0.36 \end{aligned}$$

Suppose $\hat{y} = \begin{bmatrix} 0.5 \\ 0.3 \\ 0.2 \end{bmatrix}$ & $y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

Then,

$$\begin{aligned} CE &= -\log 0.5 \\ &= 0.69 \end{aligned}$$

k Nearest Neighbors

Nearest neighbors

Algorithm: nearest neighbors

- Training: just store D_{train}
- Predictor $f(x')$:
  Find $(x, y) \in D_{\text{train}}$ where $||\phi(x) - \phi(x')||$ is smallest
  Return y

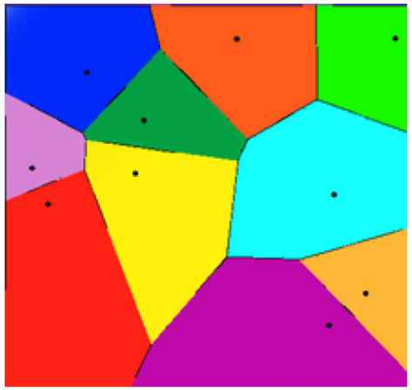
Key idea: similarity

- Similar examples tend to have similar outputs.

Expressivity of nearest neighbors

Decision boundary: based on Voronoi diagram

- Much more expressive than quadratic features
- **Non-parametric**: the hypothesis class adapts to number of examples.
- Simple and powerful, but **slow in prediction**



Summary of learners

Linear predictors: combine raw features

- prediction is fast, easy to learn, weak use of features

Neural networks: combine learned features

- prediction is fast, hard to learn, powerful use of features

Nearest neighbors: predict according to similar examples

- prediction is slow, easy to learn, powerful use of features