VIRGINIA TECH

Classification: Perceptron

Lifu Huang

Computer Science, Virginia Tech

Feb. 11, 2021

Slides adapted from Luke Zettlemoyer, David Sontag, Bert Huang

Recap - Logistic Regression

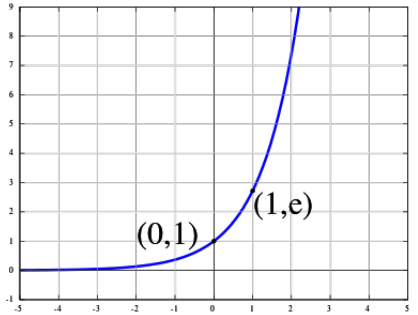
- Learn $P(Y|\boldsymbol{X})$ directly
  - Reuse ideas from regression, but let y-interpret define the probability

$$P(Y=1|\mathbf{X},\mathbf{w}) \propto exp(w_0 + \sum_i w_i X_i)$$
$$P(Y=0|\mathbf{X},\mathbf{w}) \propto 1 = \exp(0)$$

  - with normalization constants

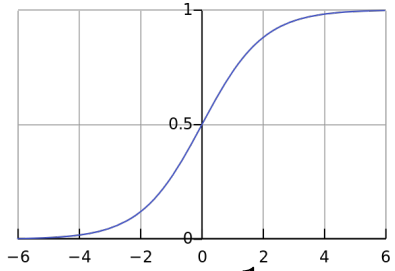
$$P(Y=0|\mathbf{X},\mathbf{w}) = \frac{1}{1+exp(w_0+\sum_i w_i X_i)}$$
$$P(Y=1|\mathbf{X},\mathbf{w}) = \frac{exp(w_0+\sum_i w_i X_i)}{1+exp(w_0+\sum_i w_i X_i)}$$

Exponential:



$$y = e^x = exp(x)$$

Logistic function:



$$y = \frac{1}{1+exp(-x)}$$

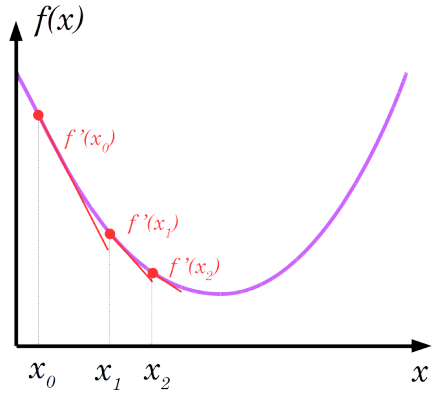
Recap - Maximize Conditional Log Likelihood

$$l(\mathbf{w}) = \sum_j y^j (w_0 + \sum_i^n w_i x_i^j) - \ln(1 + exp(w_0 + \sum_i^n w_i x_i^j))$$

$$\frac{\partial l(w)}{\partial w_i} = \sum_j x_i^j \left(y^j - P(Y^j = 1 | x^j, w)\right)$$

Gradient: direction and rate of fastest increase of the function

The **direction** of the **gradient** is the **direction** in which the function increases most quickly

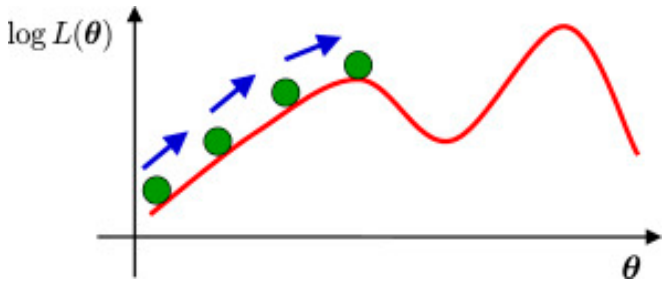


- No closed form solution to maximize $l(\boldsymbol{w})$
- Gradient Descent and Gradient Ascent !!
  - Gradient Descent: find a local minimum for a differentiable function

$$w_i^{t+1} \leftarrow w_i^t - \eta \frac{\partial l(w)}{\partial w_i}$$

  - Gradient Ascent: find a local maximum for a differentiable function

$$w_i^{t+1} \leftarrow w_i^t + \eta \frac{\partial l(w)}{\partial w_i}$$



Recap - Gradient Ascent for Logistic Regression

Gradient ascent algorithm: (learning rate $\eta > 0$)

`do:`

$$w_0^{(t+1)} \leftarrow w_0^{(t)} + \eta \sum_j [y^j - \hat{P}(Y^j = 1 \mid \mathbf{x}^j, \mathbf{w})]$$

`For i=1…n: (iterate over weights)`

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \sum_j x_i^j [y^j - \hat{P}(Y^j = 1 \mid \mathbf{x}^j, \mathbf{w})]$$

`until “change” < ε`

Loop over training examples!

Recap – K-Nearest Neighbors (KNN)

- Idea:
  - Similar examples have similar label.
  - Classify new examples like similar training examples.
- Algorithm:
  - Given some new example x for which we need to predict its class y
  - Find most similar training examples
  - Classify *x* “like” these most similar examples
- Questions:
  - How to determine similarity?
  - How many similar training examples to consider?
  - How to resolve inconsistencies among the training examples?

Who needs probabilities?

- Previously: model data with distributions
- Joint: $P(X, Y)$
  - e.g., Naïve Bayes
- Conditional: $P(Y|X)$
  - e.g., logistic regression
- But wait, why probabilities?
- Let's try to be error driven!!

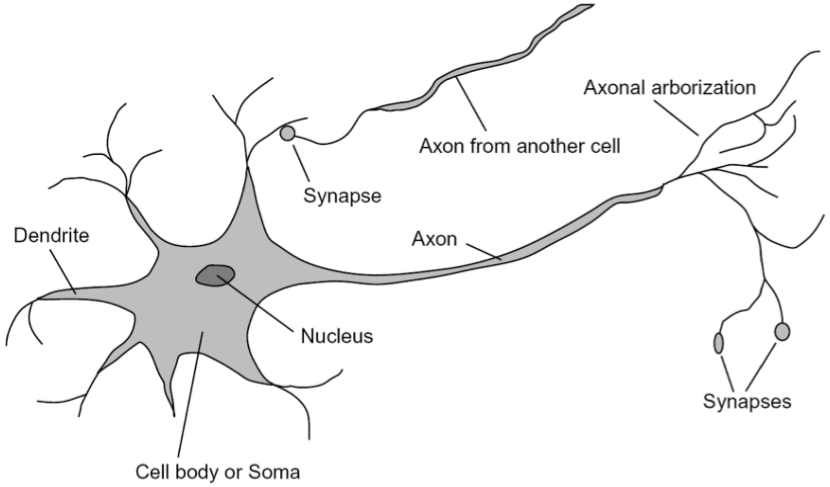
mpg	cylinders	displacement	horsepower	weight	acceleration	modelyear	maker
good	4	low	low	low	high	75to78	asia
bad	6	medium	medium	medium	medium	70to74	america
bad	4	medium	medium	medium	low	75to78	europe
bad	8	high	high	high	low	70to74	america
bad	6	medium	medium	medium	medium	70to74	america
bad	4	low	medium	low	medium	70to74	asia
bad	4	low	medium	low	low	70to74	asia
bad	8	high	high	high	low	75to78	america
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
bad	8	high	high	high	low	70to74	america
good	8	high	medium	high	high	79to83	america
bad	8	high	high	high	low	75to78	america
good	4	low	low	low	low	79to83	america
bad	6	medium	medium	medium	high	75to78	america
good	4	medium	low	low	low	79to83	america
good	4	low	low	medium	high	79to83	america
bad	8	high	high	high	low	70to74	america
good	4	low	medium	low	medium	75to78	europe
bad	5	medium	medium	medium	medium	75to78	europe

Generative vs. Discriminative

- Generative Classifiers
  - e.g., Naïve Bayes
  - A joint probability model with evidence variables
- Discriminative Classifiers
  - No generative model, no Bayes rule, often no probabilities at all
  - Try to predict the label Y directly from X
  - Robust, accurate with varied features
  - **Loosely: mistake driven rather than model driven**

Linear Classifier: Perceptron

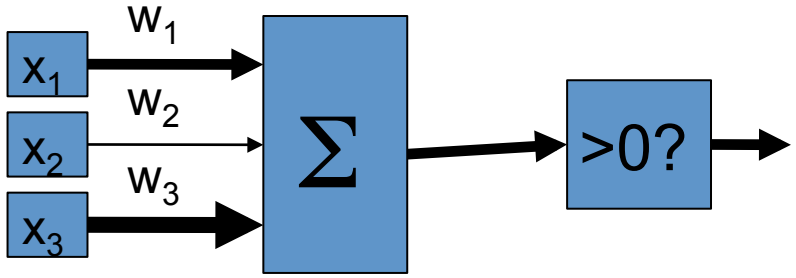
- Inputs are feature values
- Each feature has a weight
- Feed the weighted sum to activation function



$$activation_w(x) = activation\left(\sum_i w_i \cdot x_i\right) = activation(w \cdot x) = sign(w \cdot x)$$

$$\operatorname{sgn}(x) := \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

- Based on activation function, if weighted sum is:
  - Positive, output class 1
  - Negative, output class -1

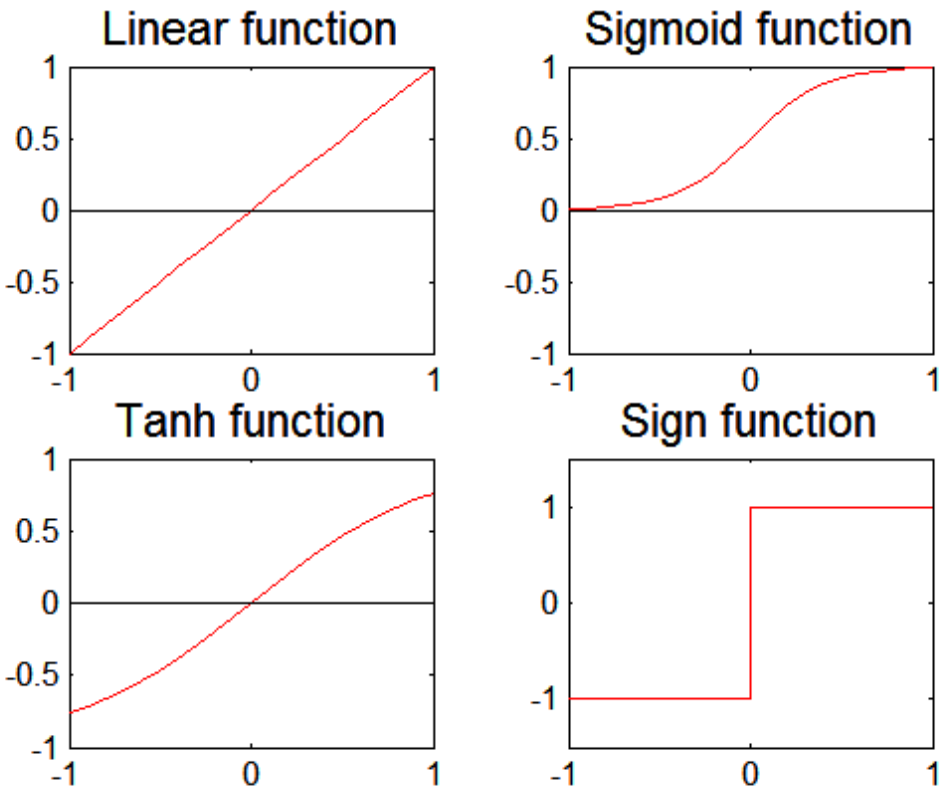


Why Activation

- Convert the output from previous layer to next
- Introduce non-linearity

$$y = f_0\left(f_1\left(f_2(\dots f_n(x))\right)\right)$$
$$= w_0 \cdot w_1 \cdot w_2 \dots w_n \cdot x = w \cdot x$$

- Linear output could be very large or small, which makes the model difficult to converge
- Activation functions: Sign, Sigmoid, Softmax, Tanh, ReLU, …



Example: Spam Filter

- Imagine 3 features (Spam is “positive” class):
  - Free (number of occurrences of “free”)
  - Money (number of occurrences of “money”)
  - BIAS (intercept, always has value 1)

“free money”

x

```
BIAS  :  1
free  :  1
money :  1
...
```

w

```
BIAS  : -3
free  :  4
money :  2
...
```

$$(1)(-3) + (1)(4) + (1)(2) + \ldots = 3$$

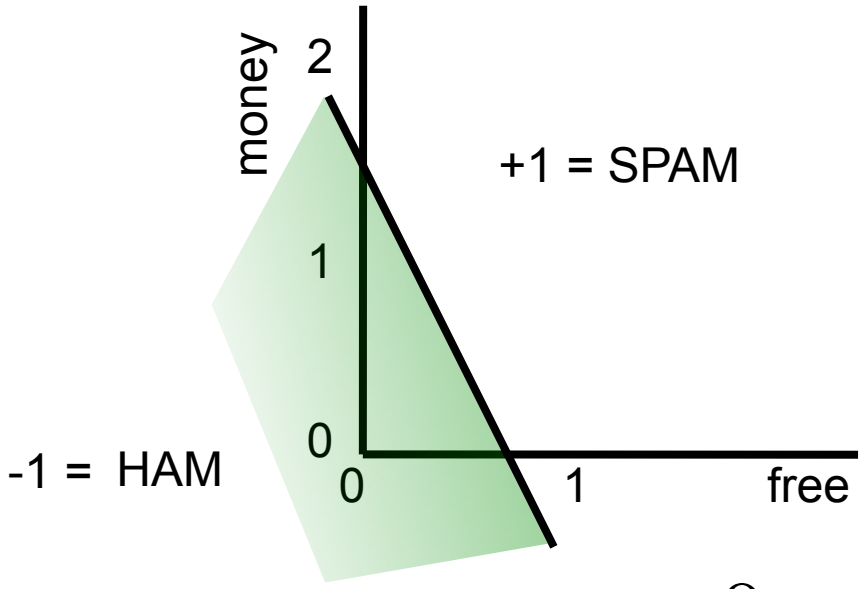
w•x > 0 ➔ SPAM!!!

Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to y=+1
  - Other corresponds to y=-1

$$w$$

```
BIAS   : -3
free   :  4
money  :  2
...
```



$$w \cdot x = 0$$

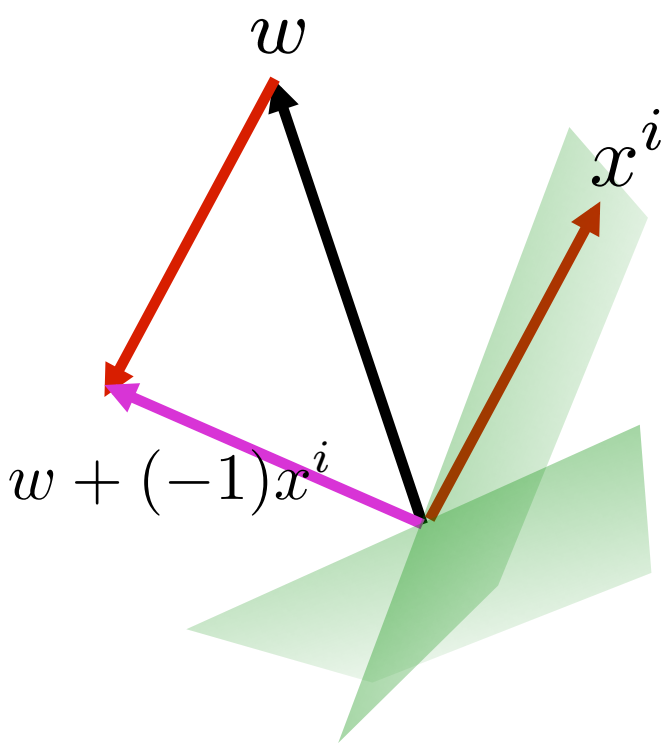
Binary Perceptron Algorithm

- Start with zero weights: $w = 0$
- For $t = 1, 2, \ldots, T$ (T passes over data)
  - For $i = 1, 2, \ldots, n$: (each training example)
    - Classify with current weights

$$y = sign(w \cdot x^i)$$

      - $sign\,(x)$ is $+1$ if w $\cdot\, x > 0$, else -1
    - If correct (i.e., $y = y^i$), no change!
    - If wrong: update

$$w = w + y^i x^i$$



Intuition of the Update Rule

- At time step *t*, for a particular instance *i:* $y^i = sign(w^t \cdot x^i)$
- If y^i is *correct*: continue
- If y^i is *incorrect*: $w^{t+1} = w^t + \hat{y}^i \cdot x^i$
- If x^i is incorrectly classified as negative, namely $\hat{y}^i = 1$, $w^t \cdot x^i < 0$:

$$x^i \cdot w^{t+1} = x^i \cdot w^t + x^i \cdot \hat{y}^i \cdot x^i = x^i \cdot w^t + \hat{y}^i \cdot (x^i \cdot x^i)$$ ↑

- If x^i is incorrectly classified as positive, namely $\hat{y}^i = -1$, $w^t \cdot x^i > 0$:

$$x^i \cdot w^{t+1} = x^i \cdot w^t + x^i \cdot \hat{y}^i \cdot x^i = x^i \cdot w^t + \hat{y}^i \cdot (x^i \cdot x^i)$$ ↓

Extended Reading: http://hagan.okstate.edu/4_Perceptron.pdf

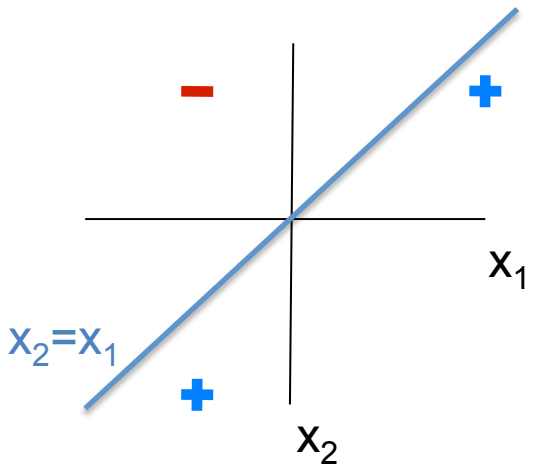
Example

$$y = sign(w \cdot x^i)$$

Update Rule:

$$w = w + y^i x^i$$

x_1	x_2	y
3	2	1
-2	2	-1
-2	-3	1



- Initial: $w = [0, 0]$
- $t = 1,\ i = 1$
  - $[0, 0] \cdot [3, 2] = 0, sign(0) = -1$
  - $w = [0, 0] + [3, 2] = [3, 2]$
- $t = 1,\ i = 2$
  - $[3, 2] \cdot [-2, 2] = -2, sign(-2) = -1$
- $t = 1,\ i = 3$
  - $[3, 2] \cdot [-2, -3] = -12, sign(-12) = -1$
  - $w = [3, 2] + [-2, -3] = [1, -1]$
- $t = 2,\ i = 1$
  - $[1, -1] \cdot [3, 2] = 1, sign(1) = 1$
- $t = 2,\ i = 2$
  - $[1, -1] \cdot [-2, 2] = -4, sign(-4) = -1$
- $t = 2,\ i = 3$
  - $[1, -1] \cdot [-2, -3] = 1, sign(1) = 1$
- Converged!!!
  - $y = w_1 x_1 + w_2 x_2 \rightarrow y = x_1 - x_2$

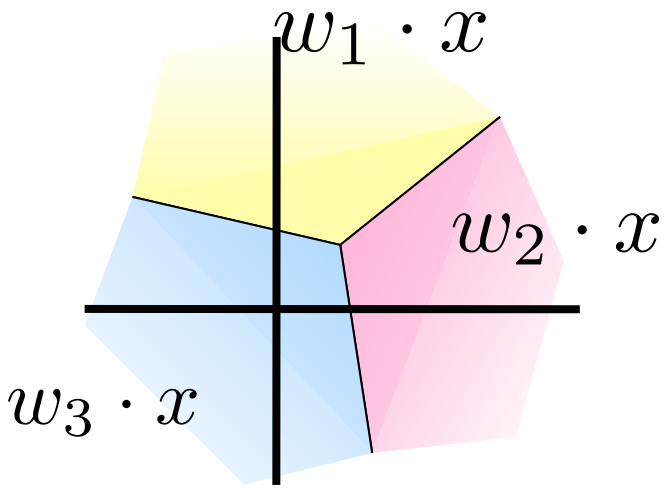
Multiclass Decision Rule

- If we have more than two classes:
  - Have a weight vector for each class w_y
  - Calculate an activation for each class

$$\text{activation}_w(x, y) = w_y \cdot x$$

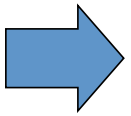
  - Highest activation wins

$$y^* = \arg\max_y(\text{activation}_w(x, y))$$



Example

"win the vote"



x

BIAS	:	1
win	:	1
game	:	0
vote	:	1
the	:	1
...		

w_{SPORTS}

BIAS	:	-2
win	:	4
game	:	4
vote	:	0
the	:	0
...		

$x \cdot w_{SPORTS} = 2$

$w_{POLITICS}$

BIAS	:	1
win	:	2
game	:	0
vote	:	4
the	:	0
...		

$x \cdot w_{POLITICS} = 7$

w_{TECH}

BIAS	:	2
win	:	0
game	:	2
vote	:	0
the	:	0
...		

$x \cdot w_{TECH} = 2$

POLITICS wins!!!

The Multi-Class Perceptron Algorithm

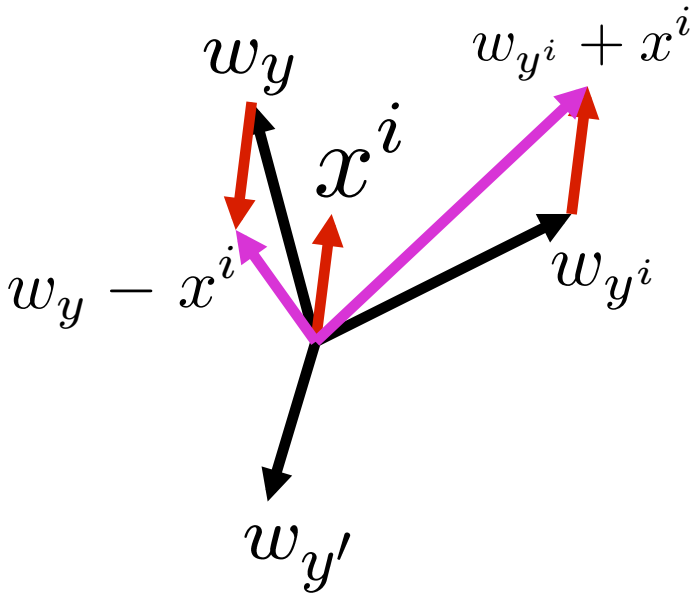
- Start with zero weights: $\forall y, w_y = 0$
- For $t = 1, 2, \ldots, T$ (T passes over data)
  - For $i = 1, 2, \ldots, n$: (each training example)
    - Classify with current weights

$$y^* = \arg\max_y (\text{activation}_w(x, y))$$

    - If correct (i.e., $y^* = y^i$), no change!
    - If wrong: subtract features x^i from weights for predicted class w_{y^*} and add them to weights for correct class w_{y_i}

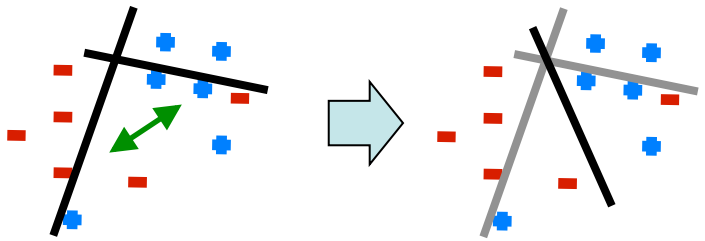
$$w_{y^*} = w_{y^*} - x^i$$

$$w_{y^i} = w_{y^i} + x^i$$



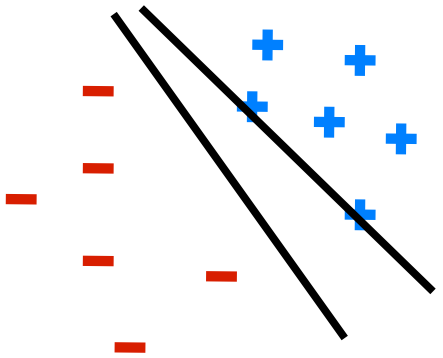
Properties of Perceptron

- Separability: some parameters get the training set perfectly correct if the data is separable
- Convergence: if the training data is separable, perceptron will eventually converge
- However …
  - If the data is not separable, weights might trash

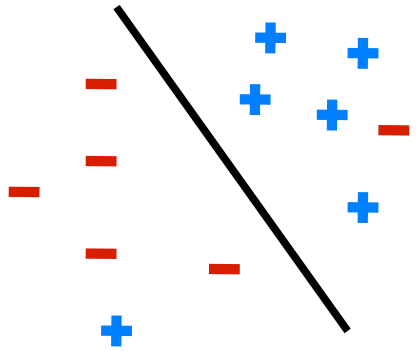


  - Mediocre generalization: finds a “barely” separating solution
  - Overtraining (overfitting): test/held-out accuracy usually arises, then falls

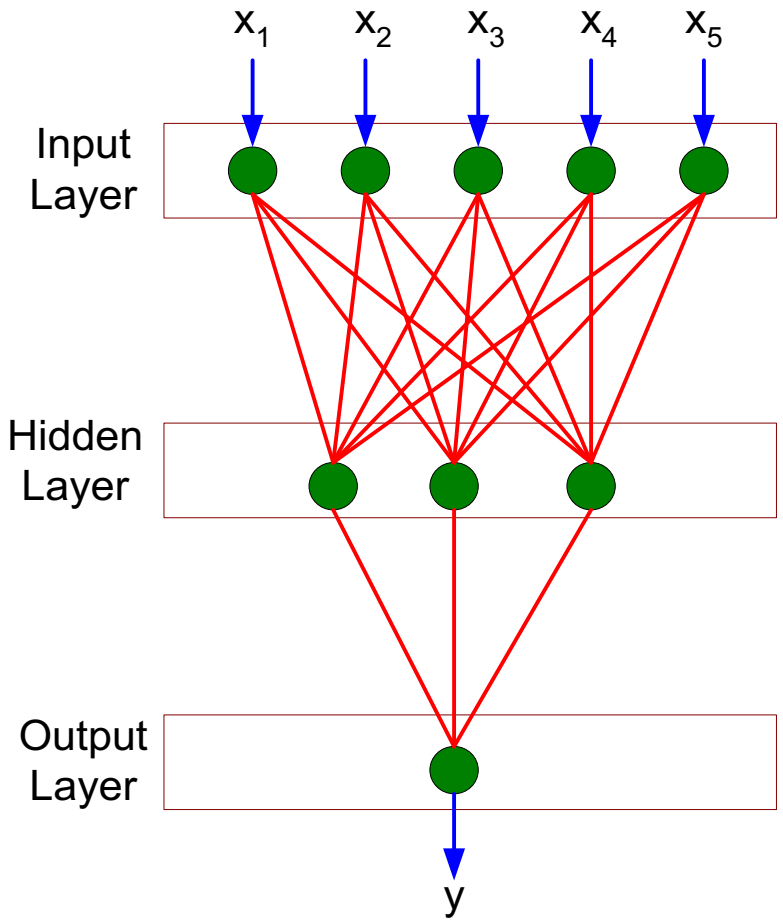
Separable



Non-Separable



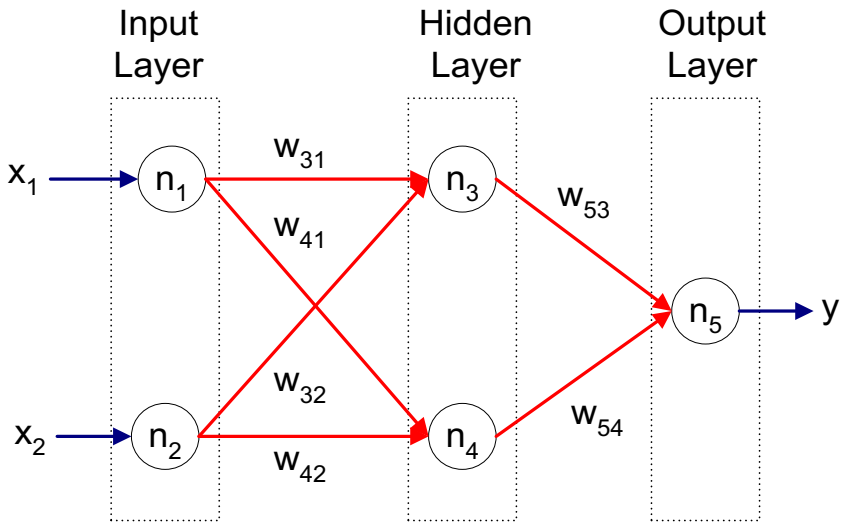
Multi-Layer Perceptron Network (MLP)

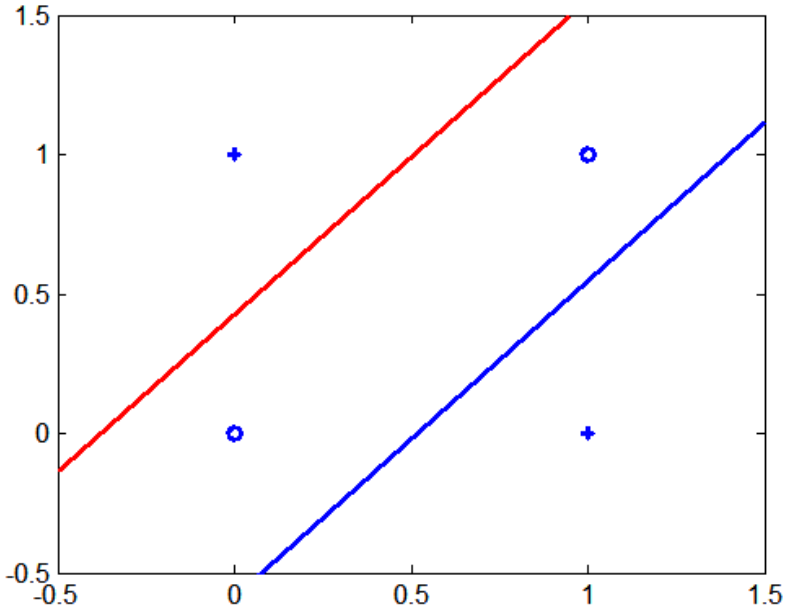


- One or more than one *hidden layer* of computing nodes
- Every node in a hidden layer operates on activations from preceding layer and transmits activations forward to nodes of next layer
- Also referred to as “feedforward neural networks” or “artificial neural networks”

Multi-Layer Perceptron Network

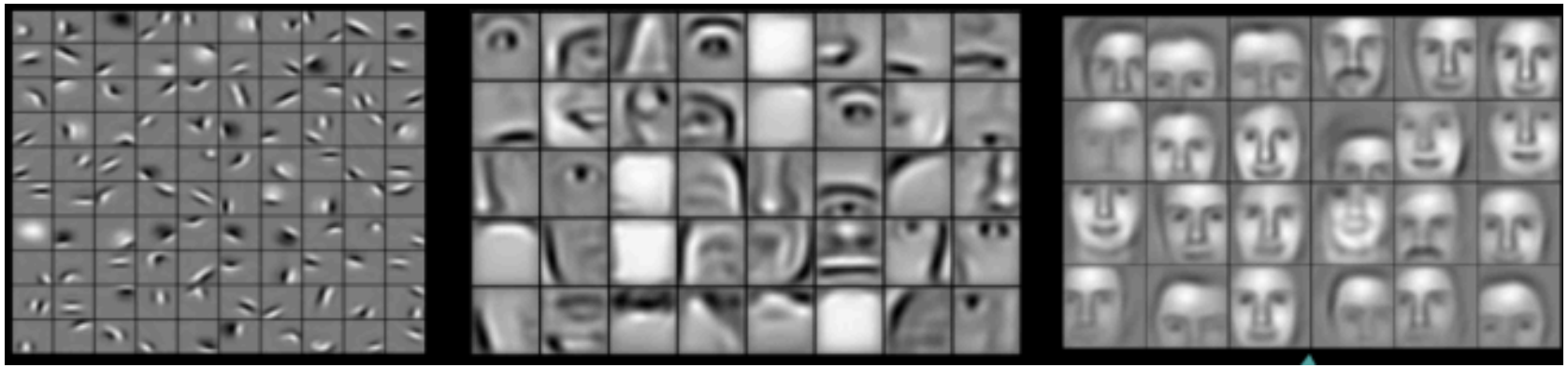
- Multi-layer neural networks with at least one hidden layer can solve any type of classification task involving nonlinear decision surfaces





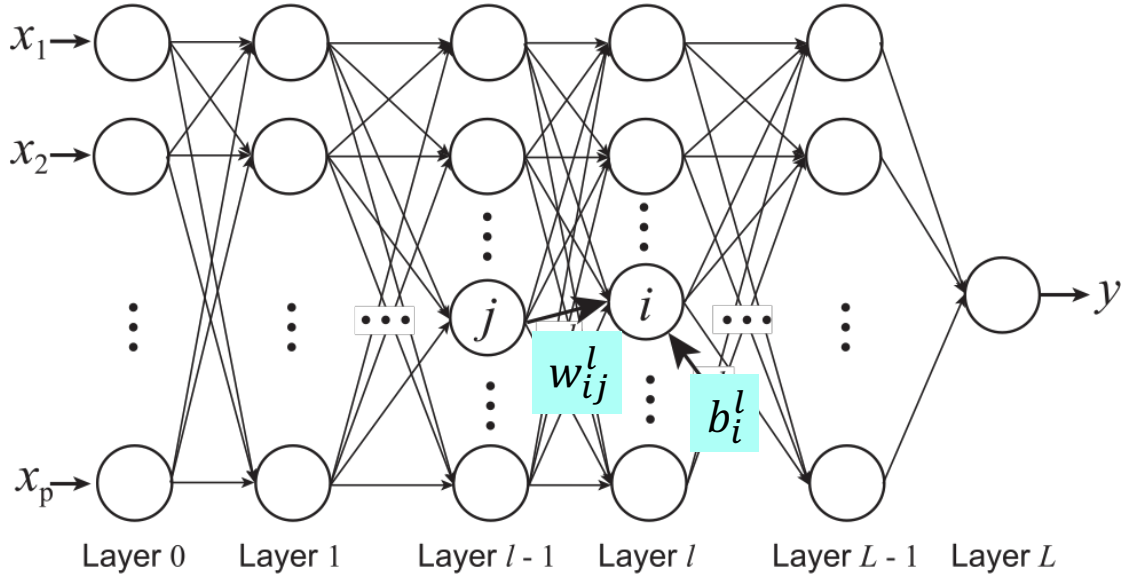
Why Multiple Hidden Layers?

- Every hidden layer represents a level of abstraction
  - *Complex features are compositions of simpler features*
  - *Activations* are used to convert the output from previous layer to next one (with certain non-linear transformation to a certain range)



- Number of layers is known as **depth** of network
  - *Deeper networks express complex hierarchy of features*

Multi-Layer Perceptron Network



$$a_i^l = f(z_i^l) = f\left(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\right)$$

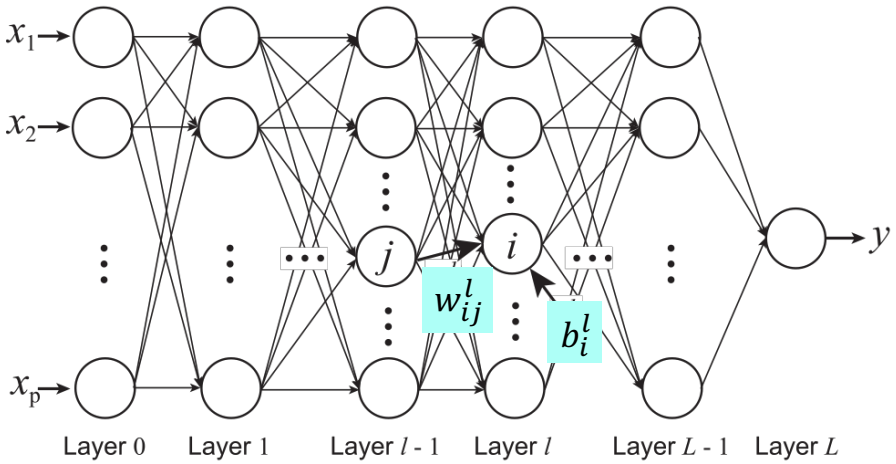
Activation value at node i at layer l

Activation Function

Linear Predictor

Learning Multi-Layer Perceptron Network

- Can we apply perceptron learning rule to each node, including hidden nodes?
  - Perceptron learning rule computes error term e = y - $\hat{y}$ and updates weights accordingly
    - Problem: how to determine the true value of y for hidden nodes?
  - Approximate error in hidden nodes by error in the output nodes
    - Problem:
      - Not clear how adjustment in the hidden nodes affect overall error
      - No guarantee of convergence to optimal solution



Gradient Descent

- Loss Function to measure errors across all training points

$$E(\mathbf{w}, \mathbf{b}) = \sum_{k=1}^{n} \text{Loss } (y_k,\ \hat{y}_k)$$

Squared Loss:

$$\text{Loss } (y_k,\ \hat{y}_k) = (y_k - \hat{y}_k)^2$$

- Gradient descent: Update parameters in the direction of "maximum descent" in the loss function across all points

$$w_{ij}^l \longleftarrow w_{ij}^l - \lambda \frac{\partial E}{\partial w_{ij}^l},$$

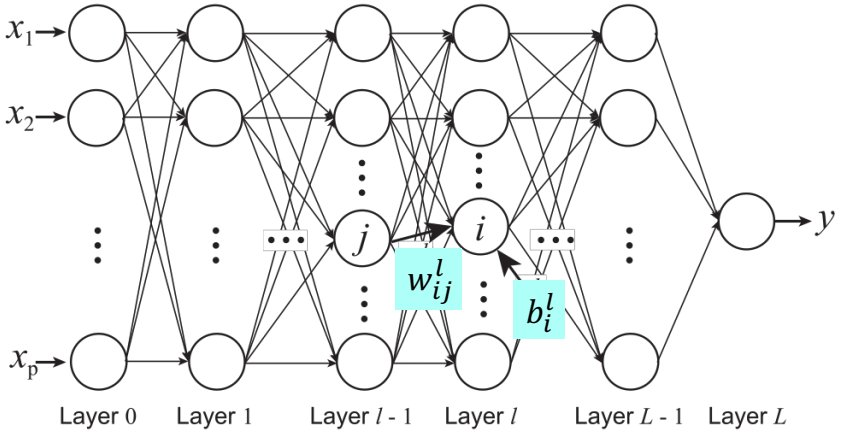
$$b_i^l \longleftarrow b_i^l - \lambda \frac{\partial E}{\partial b_i^l},$$

λ: learning rate

- Stochastic gradient descent (SGD): update the weight for every instance (minibatch SGD: update over min-batches of instances)

Computing Gradients for Layer l

$$\frac{\partial E}{\partial w_j^l} = \sum_{k=1}^{n} \frac{\partial \text{ Loss } (y_k,\ \hat{y}_k)}{\partial w_j^l}. \qquad \hat{y} = a^L$$



- Using chain rule of differentiation (on a single instance):

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \frac{\partial \text{ Loss}}{\partial a_i^l} \times \frac{\partial a_i^l}{\partial z_i^l} \times \frac{\partial z_i^l}{\partial w_{ij}^l}.$$

$$z_i^l = g(w_i^l) = \sum_j w_{ij}^l a_j^{l-1} + b_i^l$$

$$a_i^l = f(z_i^l) = f\left(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\right)$$

- For sigmoid activation function:

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \delta_i^l \times a_i^l(1 - a_i^l) \times a_j^{l-1},$$

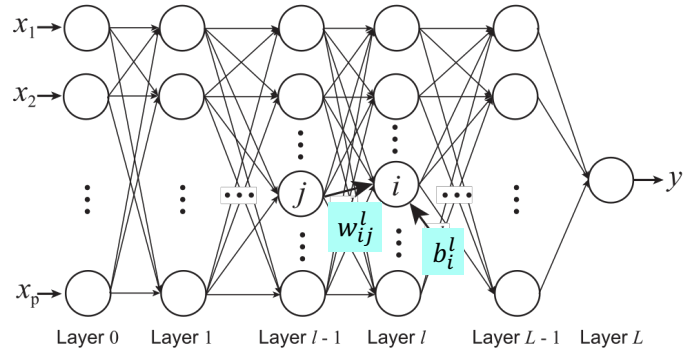
$$\text{where } \delta_i^l = \frac{\partial \text{ Loss}}{\partial a_i^l}.$$

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

Sigmoid function

- How can we compute δ_i^l for every layer?

Backpropagation Algorithm



- At output layer L:

$$\delta^L = \frac{\partial \text{ Loss}}{\partial a^L} = \frac{\partial \ (y - a^L)^2}{\partial a^L} = 2(a^L - y). \qquad \delta_j^{L-1} = \frac{\partial \ Loss}{\partial \ a_j^{L-1}} = \sum_i \frac{\partial \ Loss}{\partial \ a_i^L} \frac{\partial a_i^L}{\partial \ a_j^{L-1}}$$

- At a hidden layer l (using chain rule):

$$\delta_j^l = \sum_i (\delta_i^{l+1} \times a_i^{l+1}(1 - a_i^{l+1}) \times w_{ij}^{l+1}).$$

  - Gradients at layer l can be computed using gradients at layer l + 1
  - Start from layer L and "backpropagate" gradients to all previous layers

- Use gradient descent to update weights at every epoch
- For next epoch, use updated weights to compute loss fn. and its gradient
- Iterate until convergence (loss does not change)

Backpropagation Algorithm

Forward pass:

Backward pass

1: Let $D.train = \{(\mathbf{x}_k, y_k) \mid k = 1, 2, \ldots, n\}$ be the set of training instances.
2: Set counter $c \leftarrow 0$
3: Initialize the weight and bias terms $(\mathbf{w}^{(0)}, \mathbf{b}^{(0)})$ with random values.
4: **repeat**
5: **for** each training instance $(\mathbf{x}_k, y_k) \in D.train$ **do**
6: Compute the set of activations $(a_i^l)_k$ by making a forward pass using $\mathbf{x}_k$.
7: Compute the set $(\delta_i^l)_k$ by back-propagation
8: Compute $(\partial \text{ Loss}/\partial w_{ij}^l,\ \partial \text{ Loss}/\partial b_i^l)_k$
9: **end for**
10: Compute $\partial E/\partial w_{ij}^l \longleftarrow \sum_{k=1}^n (\partial \text{ Loss}/\partial w_{ij}^l)_k$.
11: Compute $\partial E/\partial b_i^l \longleftarrow \sum_{k=1}^n (\partial \text{ Loss}/\partial b_i^l)_k$.
12: Update $(\mathbf{w}^{(c+1)}, \mathbf{b}^{(c+1)})$ by gradient descent
13: Update $c \leftarrow c + 1$.
14: **until** $(\mathbf{w}^{(c+1)}, \mathbf{b}^{(c+1)})$ and $(\mathbf{w}^{(c)}, \mathbf{b}^{(c)})$ converge to the same value.

Multi-Layer Perceptron Network

- Provides a natural way to represent a hierarchy of features at multiple levels of abstraction
  - represents complex high-level features as compositions of simpler, lower-level features
  - The greater the number of hidden layers, the deeper is the hierarchy of features learned by the network
  - This motivates the learning of MLP models with long chains of hidden layers
- Design Issues
  - Number of hidden layers and nodes per layer
  - Initial weights and biases
  - Learning rate, max. number of epochs, mini-batch size for mini-batch SGD, …

Characteristics of MLP

- Multilayer ANN are universal approximators but could suffer from overfitting if the network is too large
- Gradient descent may converge to local minimum
- Model building can be very time consuming, but testing can be very fast
- Can handle redundant and irrelevant attributes (in small numbers) because weights are automatically learnt for all attributes
- Sensitive to noise in training data
- Difficult to handle missing attributes

Deep Learning Trends

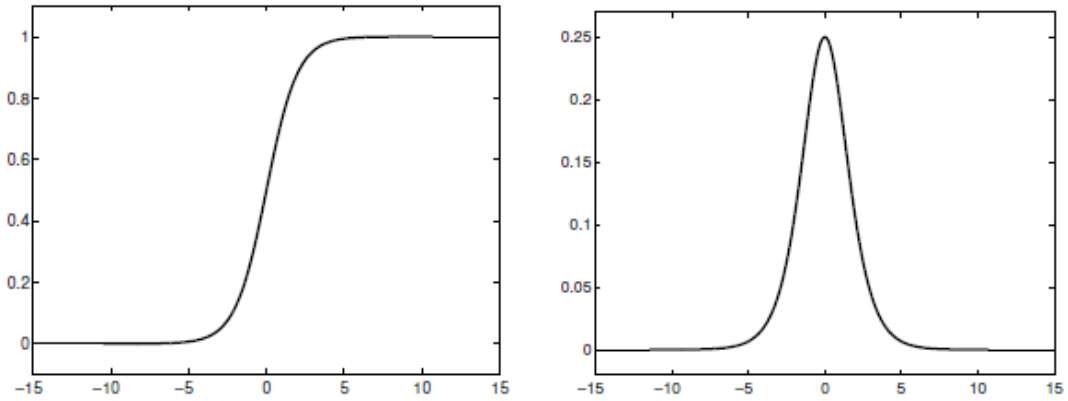
- Training **deep** neural networks (more than 5-10 layers) could only be possible in recent times with:
  - Faster computing resources (GPU)
  - Larger labeled training sets
  - **Algorithmic Improvements in Deep Learning**
- Recent Trends:
  - Specialized ANN Architectures:
    - Convolutional Neural Networks (for image data)
    - Recurrent Neural Networks (for sequence data)
    - Residual Networks (with skip connections)
  - Unsupervised Models: Autoencoders
  - Generative Models: Generative Adversarial Networks

Saturation of Output Problem

- Gradient descent:

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \frac{\partial \text{ Loss}}{\partial a_i^l} \times \frac{\partial a_i^l}{\partial z_i^l} \times \frac{\partial z_i^l}{\partial w_{ij}^l}.$$

$$\frac{\partial a_i^l}{\partial z_i^l} = \frac{\partial \, \sigma(z_i^l)}{\partial z_i^l} = a_i^l(1 - a_i^l)$$



(a) $\sigma(z)$. (b) $\partial\sigma(z)/\partial z$.

Saturation: when input to the sigmoid is very large or small, the output will close to 1 or 0, so the gradient will diminish to 0

$$\frac{\partial \text{ Loss}}{\partial w_j^L} = 2(a^L - y) \times \sigma(z^L)(1 - \sigma(z^L)) \times a_j^{L-1}. \longrightarrow$$ 0 as $\sigma \to 0$ or $\sigma \to 1$ even when $a^L \neq y$

Problem arises due to combination of loss function and activation function

Replacing Loss with Cross-Entropy

- Cross-entropy loss function

$$Loss\ (y,\ \hat{y}) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y}).$$

$$\begin{aligned}\delta^L = \frac{\partial\ Loss}{\partial a^L} &= \frac{-y}{a^L} + \frac{(1-y)}{(1-a^L)}. \\ &= \frac{(a^L - y)}{a^L(1-a^L)}.\end{aligned}$$

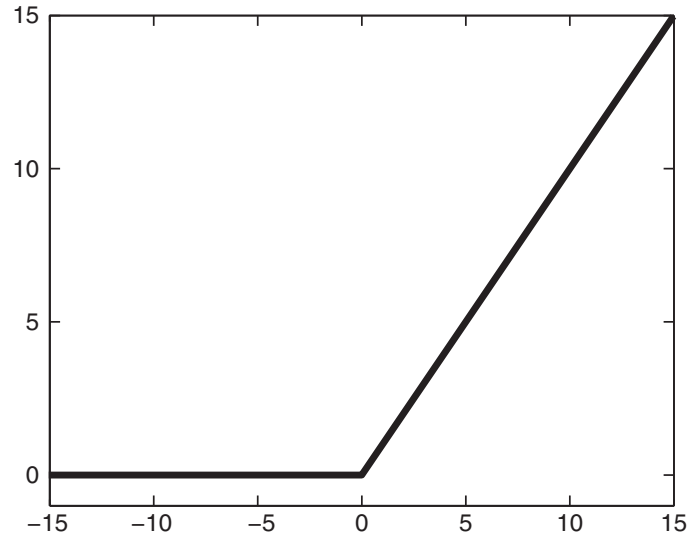
$$\begin{aligned}\frac{\partial\ \text{Loss}}{\partial w_j^L} &= \frac{(a^L - y)}{a^L(1-a^L)} \times a^L(1-a^L) \times a_j^{L-1}. \\ &= (a^L - y) \times a_j^{L-1}.\end{aligned}$$

Gradient depends on classification error; helps alleviate the saturation of output problem at the output layer

Rectified Linear Units (ReLU)

- To overcome the vanishing gradient problem, use the rectified linear unit activation function

$$a = f(z) = \begin{cases} z, & \text{if } z > 0. \\ 0, & \text{otherwise.} \end{cases}$$



  - Gradient:

$$\frac{\partial a}{\partial z} = \begin{cases} 1, & \text{if } z > 0. \\ 0, & \text{if } z < 0. \end{cases}$$

  - Although not differentiable at z=0, convention is to assume

$$\therefore \partial a / \partial z = 0$$

Avoiding Vanishing Gradients using ReLU

- Sigmoid activation function:

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \delta_i^l \times a_i^l(1 - a_i^l) \times a_j^{l-1},$$

$$= \sum_i (\delta_i^{l+1} \times a_i^{l+1}(1 - a_i^{l+1}) \times w_{ij}^{l+1}).$$

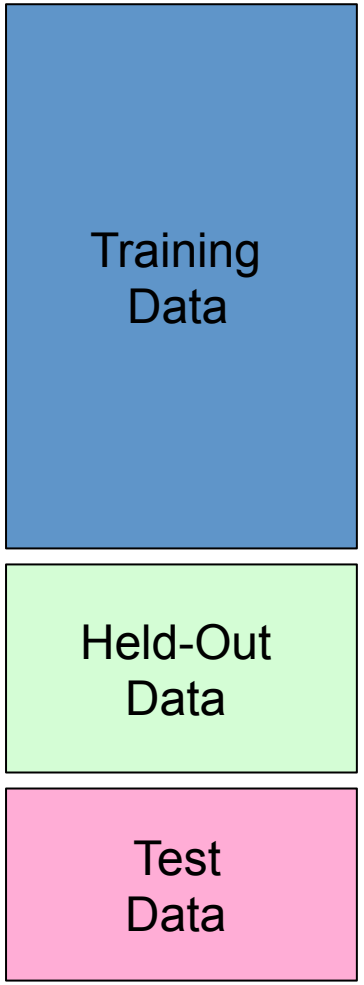
- ReLU:

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \delta_i^l \times I(z_i^l) \times a_j^{l-1},$$

$$\text{where } \delta_i^l = \sum_i (\delta_i^{l+1} \times I(z_i^{l+1}) \times w_{ij}^{l+1}),$$

$$\text{and } I(z) = \begin{cases} 1, & \text{if } z > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Three Views of Classification



- Naïve Bayes
  - Parameters from data statistics
  - Parameters: probabilistic interpretation
  - Training: one pass through the data
- Logistic Regression
  - Parameters from gradient ascent
  - Parameters: linear, probabilistic model, and discriminative
  - Training: gradient ascent, regularize to stop overfitting
- Perceptron
  - Parameters from reactions to mistakes
  - Parameters: discriminative interpretation
  - Training: go through the data until held-out accuracy maxes out