Lecture 8. Perceptron & ANNs

COMP90051 Statistical Machine Learning

Semester 1, 2021
Lecturer: Trevor Cohn

THE UNIVERSITY OF MELBOURNE

This lecture

- Perceptron
  - Introduction to Artificial Neural Networks
  - The perceptron model
  - Stochastic gradient descent
- Multiple layer networks
  - Model structure
  - Universal approximation
  - Training preliminaries

The Perceptron Model

A building block for artificial neural networks, yet another linear classifier

Biological inspiration

- Humans perform well at many tasks that matter
- Originally neural networks were an attempt to mimic the human brain

photo: Alvesgaspar,
Wikimedia Commons, CC3

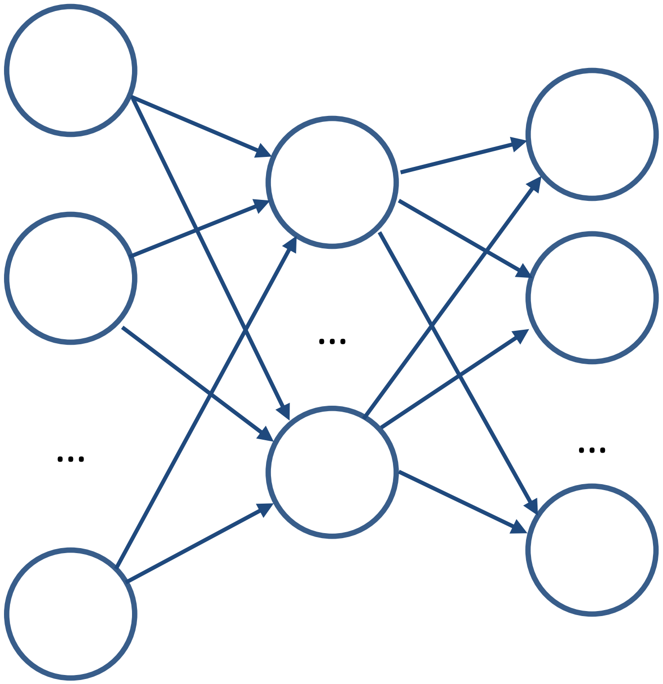
Artificial neural network

- As a *crude approximation*, the human brain can be thought as a mesh of interconnected processing nodes (neurons) that relay electrical signals

- Artificial neural network is a network of processing elements

- Each element converts inputs to output

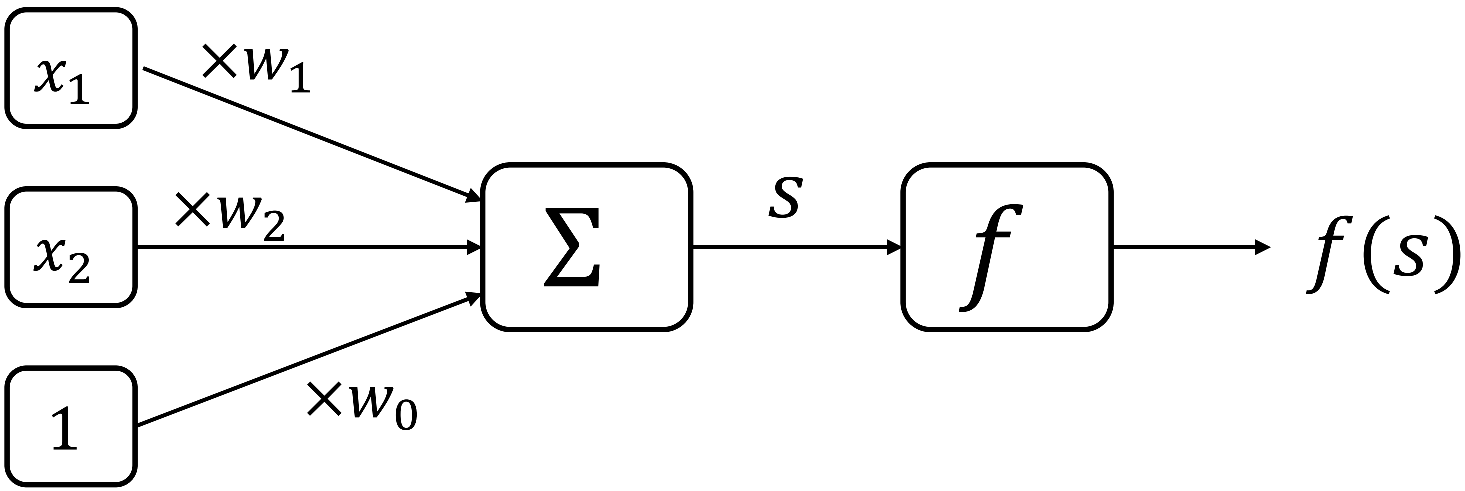
- The output is a function (called *activation function*) of a weighted sum of inputs



Outline

- In order to use an ANN we need (a) to design network topology and (b) adjust weights to given data
  - In this subject, we will exclusively focus on task (b) for a particular class of networks called feed forward networks
- Training an ANN means adjusting weights for training data given a pre-defined network topology
- First we will turn our attention to an individual network element, before building deeper architectures

Perceptron model



Compare this model to logistic regression

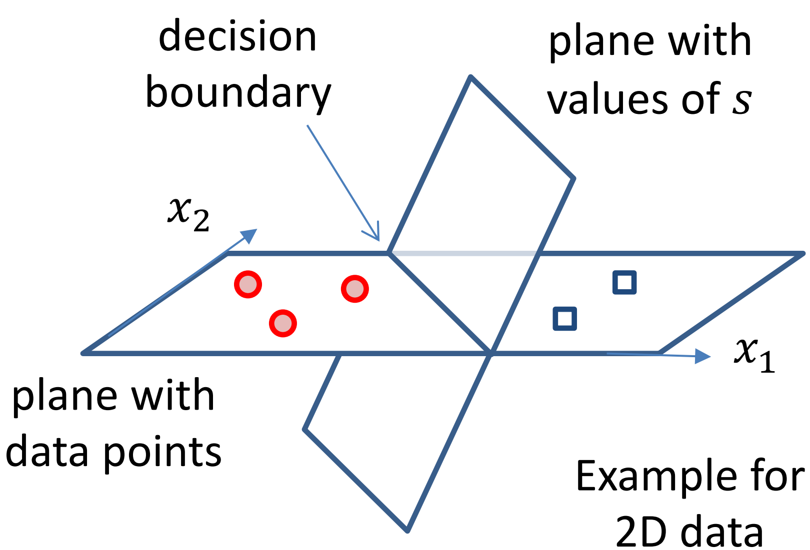
- x_1, x_2 – inputs
- w_1, w_2 – synaptic weights
- w_0 – bias weight
- f – activation function

Perceptron is a linear binary classifier

Perceptron is a binary classifier:

Predict class A if $s \geq 0$
Predict class B if $s < 0$
where $s = \sum_{i=0}^{m} x_i w_i$

Perceptron is a <u>linear classifier</u>: s is a linear function of inputs, and the decision boundary is linear

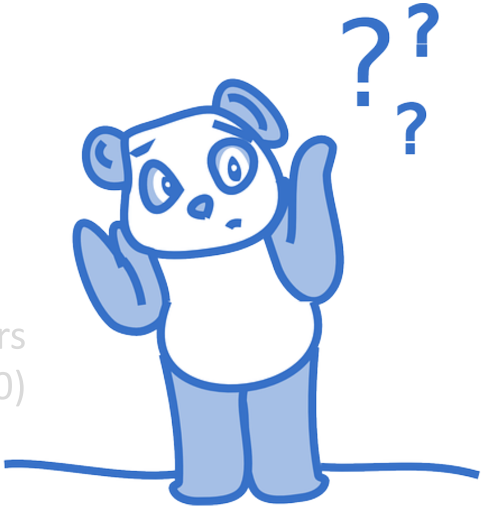


Example for 2D data

Exercise: find weights of a perceptron capable of perfect classification of the following dataset

x_1	x_2	y
0	0	Class B
0	1	Class B
1	0	Class B
1	1	Class A

art: OpenClipartVectors at pixabay.com (CC0)



Loss function for perceptron

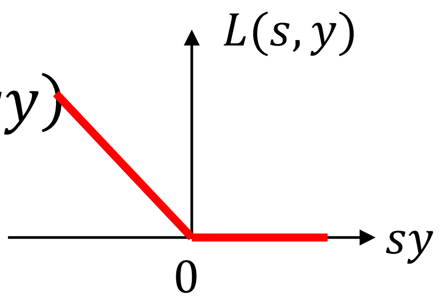
- "Training": finds weights to minimise some loss. Which?
- Our task is binary classification. Encode one class as $+1$ and the other as -1. So each training example is now $\{\boldsymbol{x}, y\}$, where y is either $+1$ or -1
- Recall that, in a perceptron, $s = \sum_{i=0}^{m} x_i w_i$, and the sign of s determines the predicted class: $+1$ if $s > 0$, and -1 if $s < 0$
- Consider a single training example.
  * If y and s have same sign then the example is classified correctly.
  * If y and s have different signs, the example is misclassified

Loss function for perceptron

- The perceptron uses a loss function in which there is no penalty for correctly classified examples, while the penalty (loss) is equal to s for misclassified examples*

- Formally:
  - $L(s, y) = 0$ if both s, y have the same sign
  - $L(s, y) = |s|$ if both s, y have different signs

- This can be re-written as $L(s, y) = \max(0, -sy)$



* This is similar, but not identical to another widely used loss function called ***hinge loss***

Stochastic gradient descent

- Randomly shuffle/split all training examples in B batches
- Choose initial $\boldsymbol{\theta}^{(1)}$
- For i from 1 to T
- For j from 1 to B
- Do gradient descent update using data from batch j

Iterations over the entire dataset are called *epochs*

- Advantage of such an approach: computational feasibility for large datasets

Perceptron training algorithm

Choose initial guess $\boldsymbol{w}^{(0)}$, $k = 0$

For i from 1 to T (epochs)

For j from 1 to N (training examples)

Consider example $\{\boldsymbol{x}_j, y_j\}$

Update*: $\boldsymbol{w}^{(k++)} = \boldsymbol{w}^{(k)} - \eta \boldsymbol{\nabla} L(\boldsymbol{w}^{(k)})$

$$L(\boldsymbol{w}) = \max(0, -sy)$$

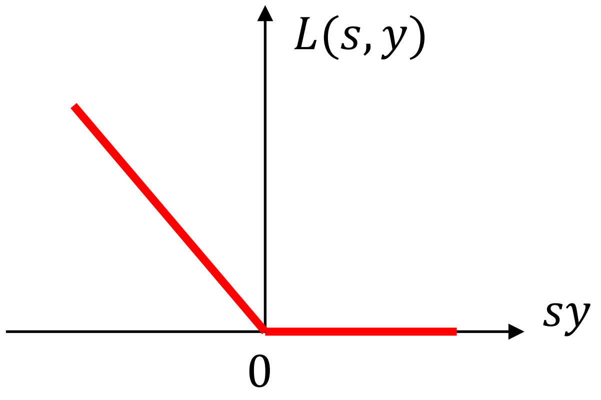
$$s = \sum_{i=0}^{m} x_i w_i$$

η is learning rate

*There is no derivative when $s = 0$, but this case is handled explicitly in the algorithm, see next slides

Perceptron training rule

- We have $\frac{\partial L}{\partial w_i} = 0$ when $sy > 0$
  - We don't need to do update when an example is correctly classified
- We have $\frac{\partial L}{\partial w_i} = -x_i$ when $y = 1$ and $s < 0$
- We have $\frac{\partial L}{\partial w_i} = x_i$ when $y = -1$ and $s > 0$



- $s = \sum_{i=0}^{m} x_i w_i$

Perceptron training algorithm

When classified correctly, weights are unchanged

When misclassified: $\boldsymbol{w}^{(k+1)} = -\eta(\pm\boldsymbol{x})$
($\eta > 0$ is called *learning rate)*

<u>If $y = 1$, but $s < 0$</u>
$w_i \leftarrow w_i + \eta x_i$
$w_0 \leftarrow w_0 + \eta$

<u>If $y = -1$, but $s \geq 0$</u>
$w_i \leftarrow w_i - \eta x_i$
$w_0 \leftarrow w_0 - \eta$

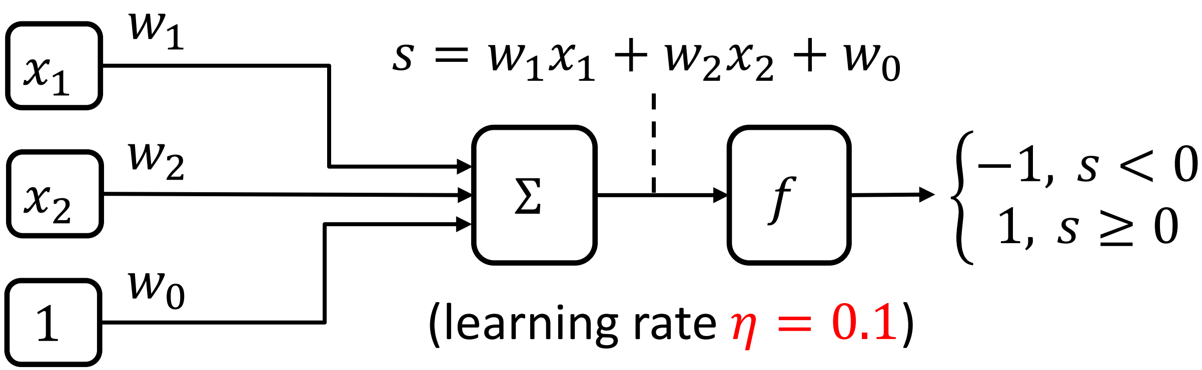
<u>*Convergence Theorem*</u>: if the training data is linearly separable, the algorithm is guaranteed to converge to a solution. That is, there exist a finite K such that $L(\boldsymbol{w}^K) = 0$

Pros and cons of perceptron learning

- If the data is linearly separable, the perceptron training algorithm will converge to a correct solution
  * There is a formal proof ← good!
  * It will converge to some solution (separating boundary), one of infinitely many possible ← bad!
- However, if the data is not linearly separable, the training will fail completely rather than give some approximate solution
  * Ugly ☹

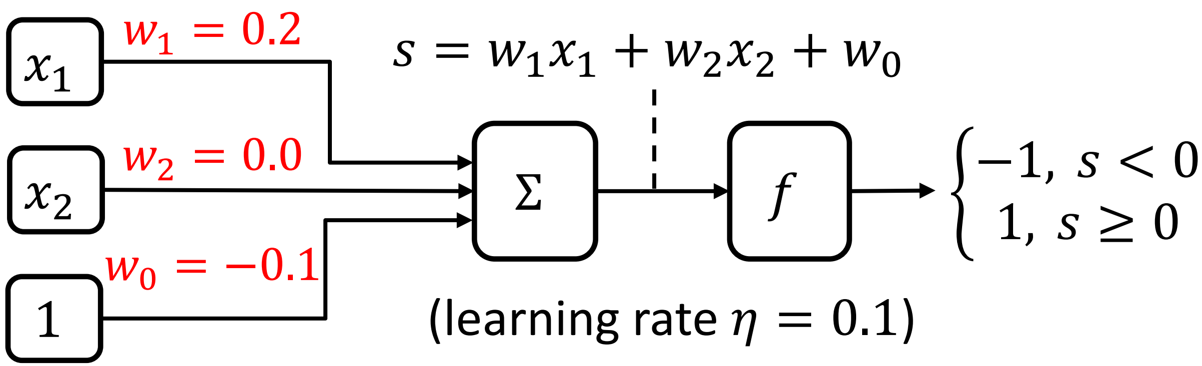
Perceptron learning example

Basic setup

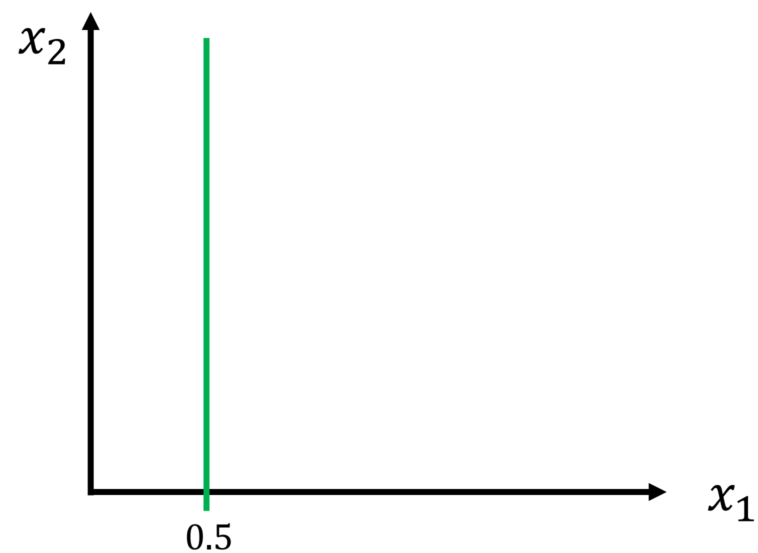


Perceptron learning example

Start with random weights

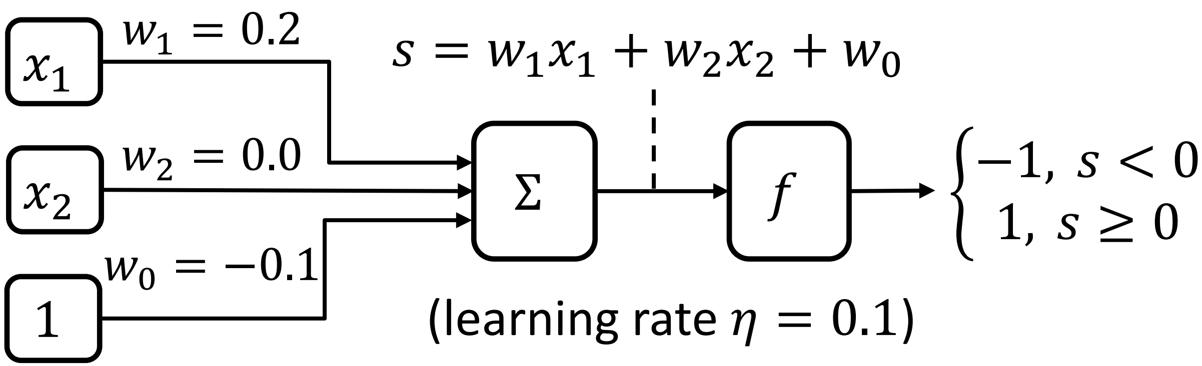


(learning rate $\eta = 0.1$)



Perceptron learning example

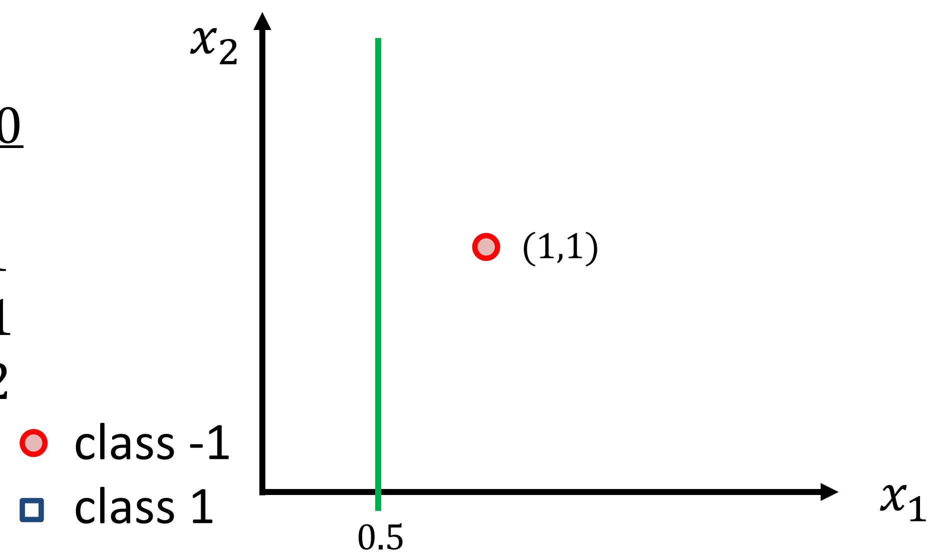
Consider training example 1



(learning rate $\eta = 0.1$)

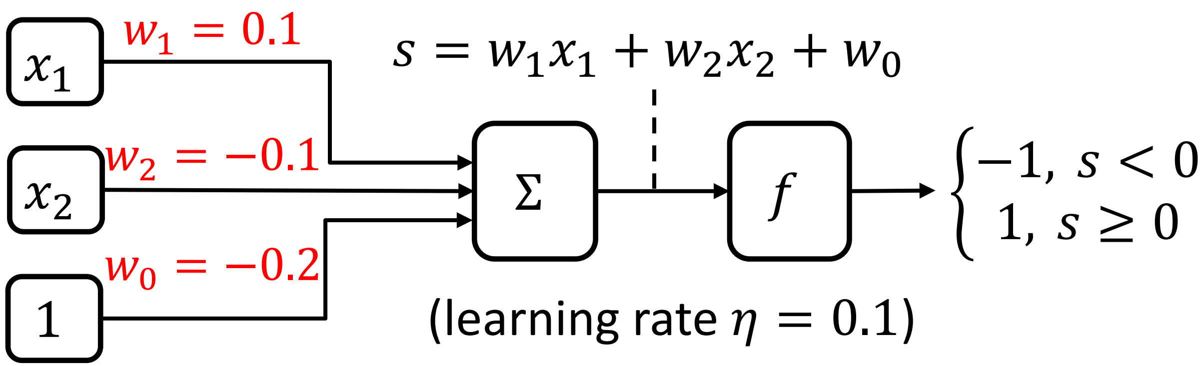
$0.2x_1 + 0.0x_2 - 0.1 > 0$

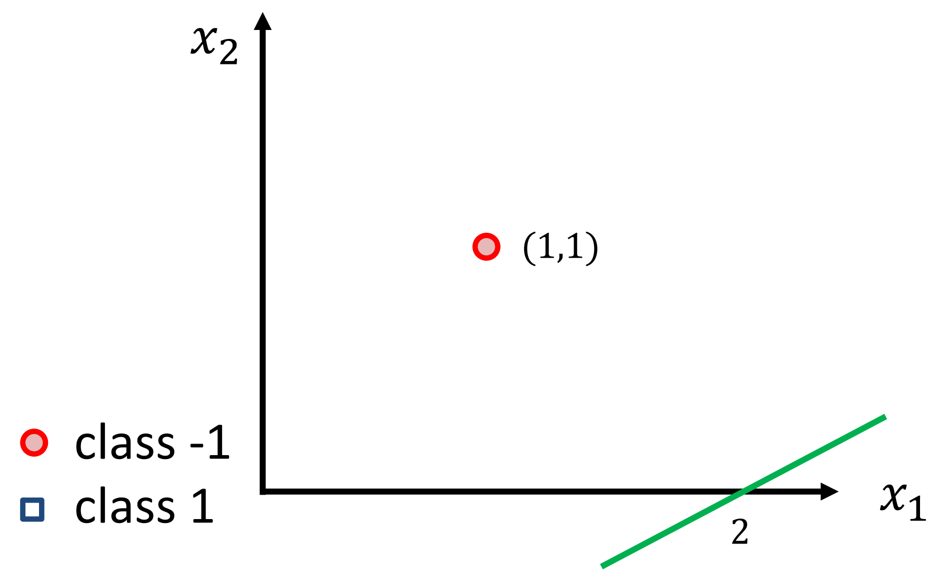
$$
\begin{aligned}
w_1 &\leftarrow w_1 - \eta x_1 = 0.1 \\
w_2 &\leftarrow w_2 - \eta x_2 = -0.1 \\
w_0 &\leftarrow w_0 - \eta = -0.2
\end{aligned}
$$



Perceptron learning example

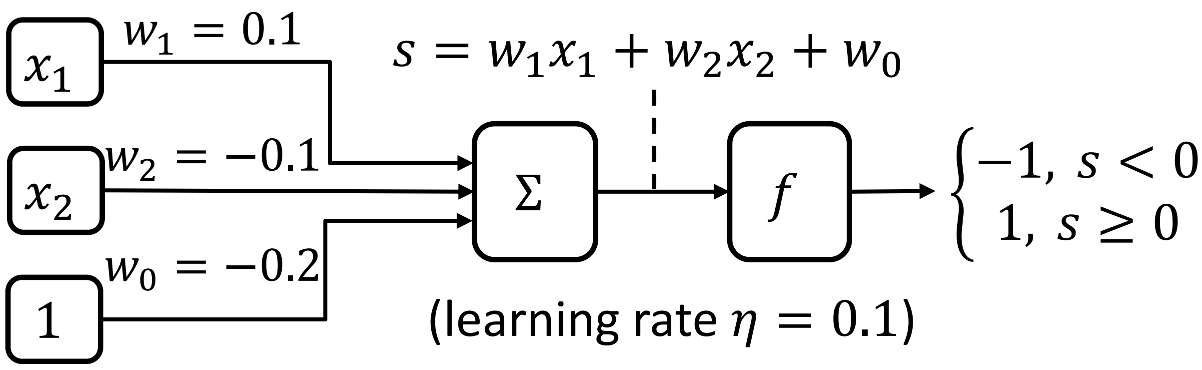
Update weights





Perceptron learning example

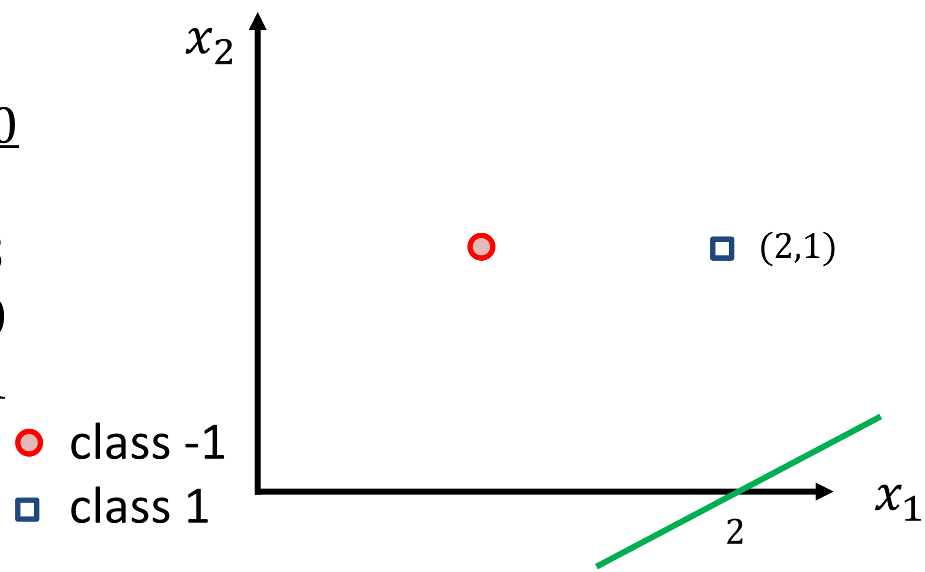
Consider training example 2



(learning rate $\eta = 0.1$)

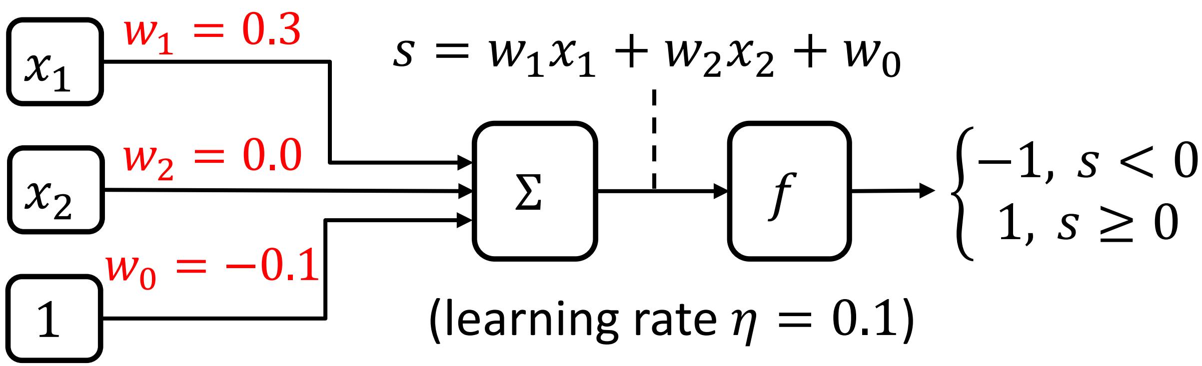
$\underline{0.1x_1 - 0.1x_2 - 0.2 < 0}$

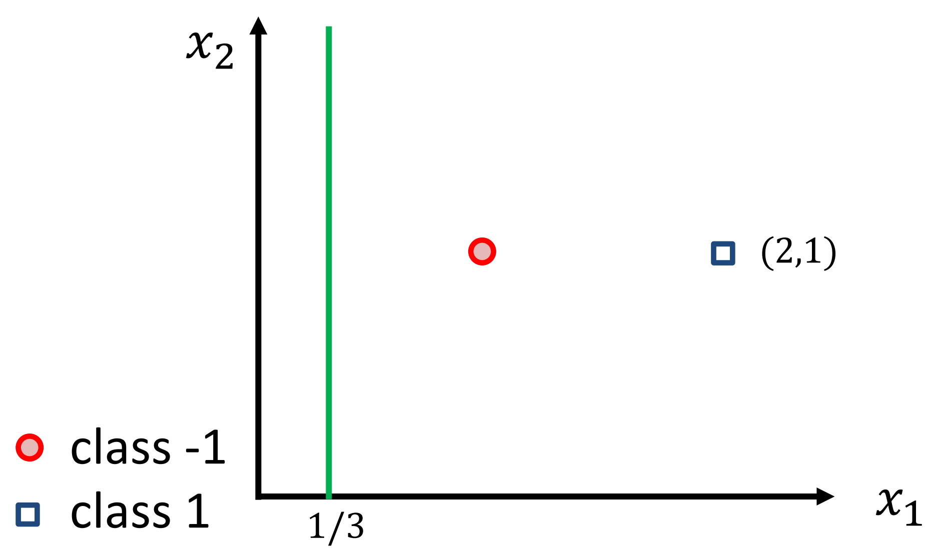
$$\begin{aligned} w_1 &\leftarrow w_1 + \eta x_1 = 0.3 \\ w_2 &\leftarrow w_2 + \eta x_2 = 0.0 \\ w_0 &\leftarrow w_0 + \eta \quad\;\; = -0.1 \end{aligned}$$



Perceptron learning example

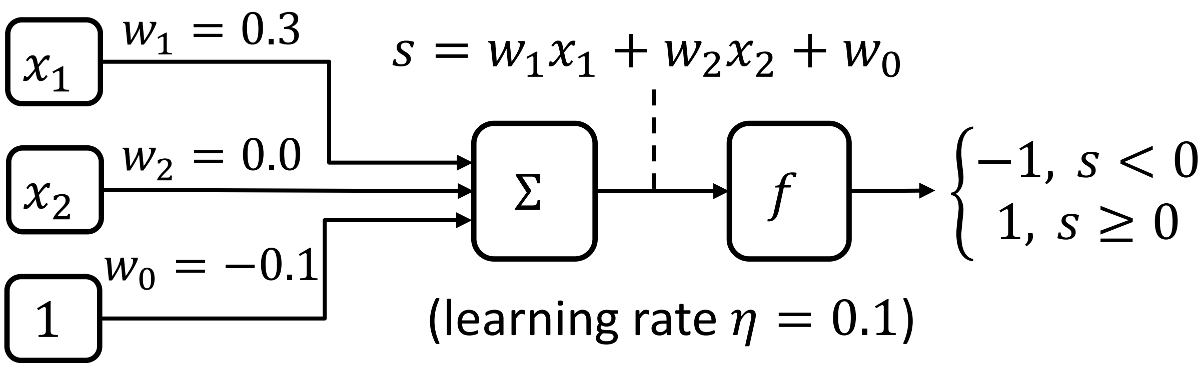
Update weights





Perceptron learning example

Further examples

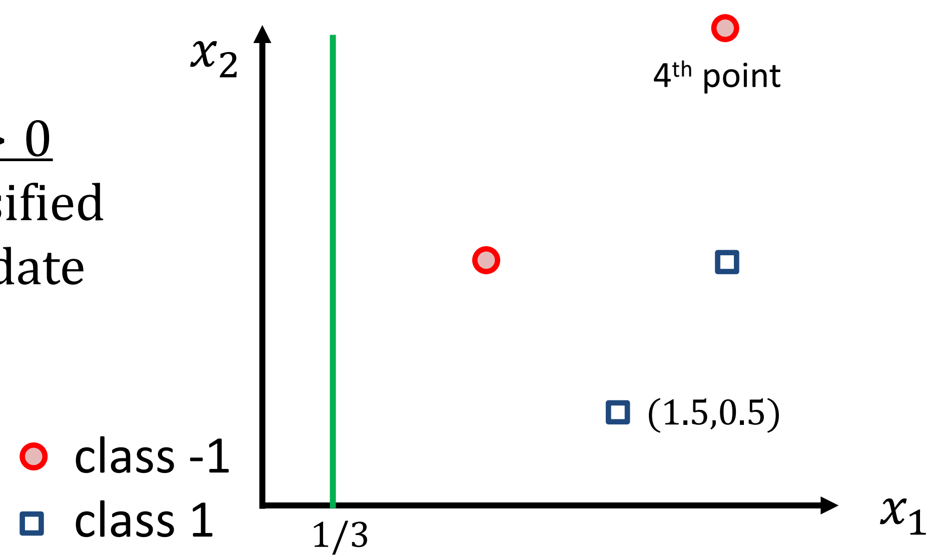


$\underline{0.3x_1 - 0.0x_2 - 0.1 > 0}$

3rd point: correctly classified

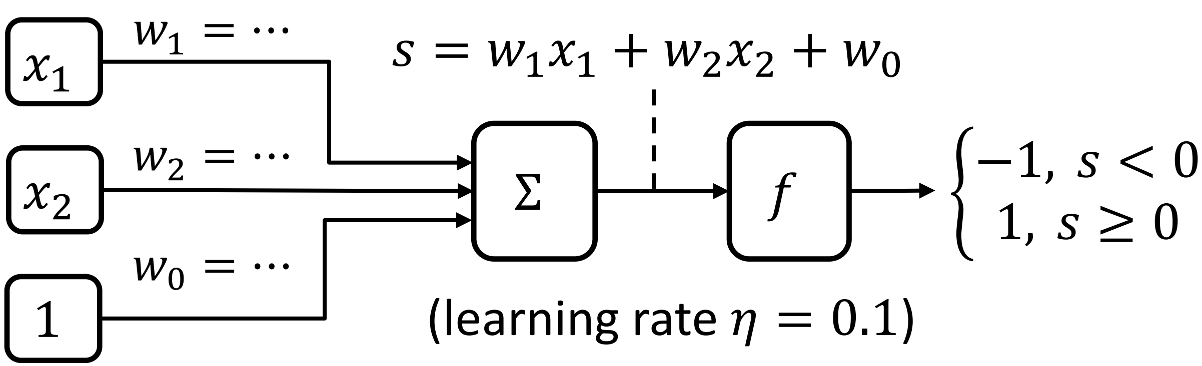
4th point: incorrect, update

etc.

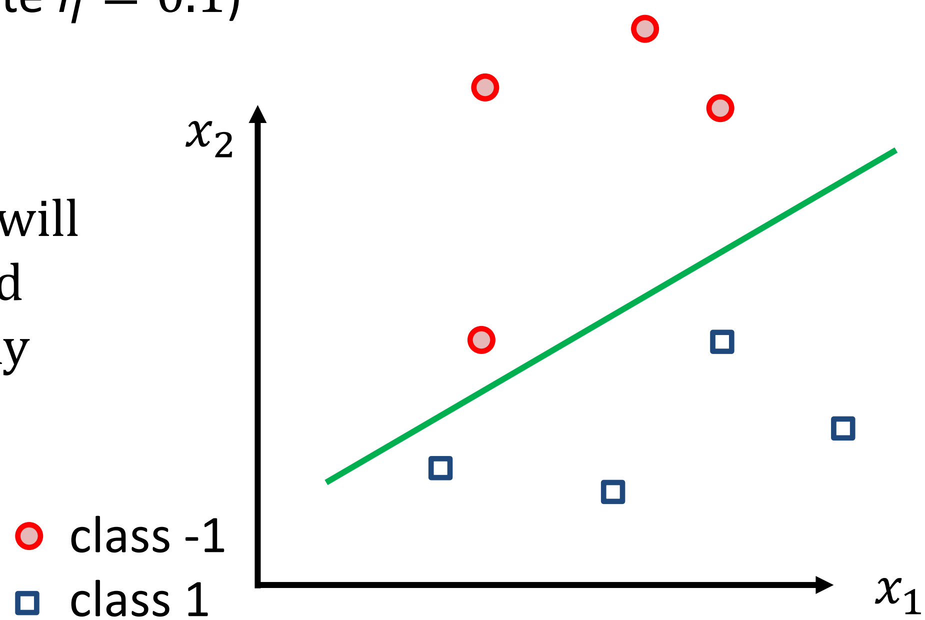


Perceptron learning example

Further examples



Eventually, all the data will be correctly classified (provided it is linearly separable)



Mini Summary

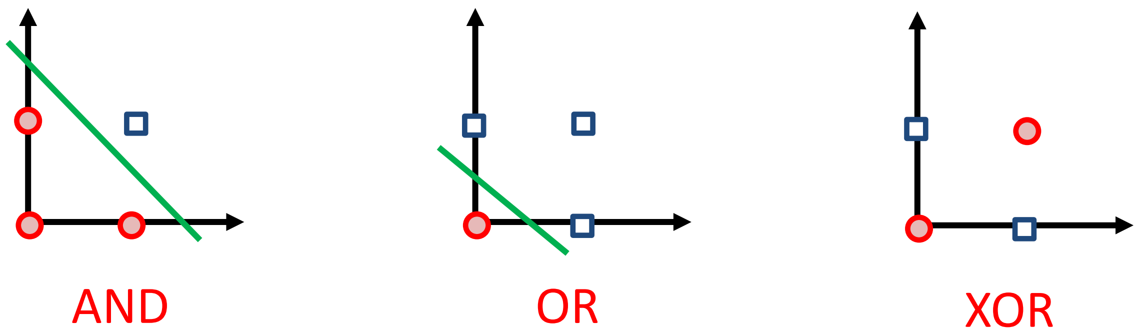
- Perceptron
  - Introduction to Artificial Neural Networks
  - The perceptron model
  - Stochastic gradient descent

Multilayer Perceptron

Modelling non-linearity via
function composition

Limitations of linear models

Some problems are linearly separable, but many are not



Possible solution: composition

$x_1 \text{ XOR } x_2 = (x_1 \text{ OR } x_2) \text{ AND not}(x_1 \text{ AND } x_2)$

We are going to compose perceptrons …

Perceptron is *sort of* a building block for ANN

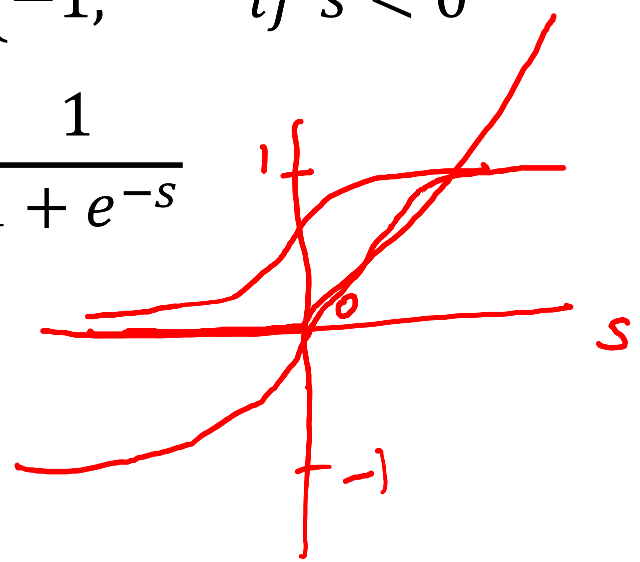
- ANNs are not restricted to binary classification
- Nodes in ANN can have various activation functions

Step function $$f(s) = \begin{cases} 1, & if\ s \geq 0 \\ 0, & if\ s < 0 \end{cases}$$

Sign function $$f(s) = \begin{cases} 1, & if\ s \geq 0 \\ -1, & if\ s < 0 \end{cases}$$

Logistic function $$f(s) = \frac{1}{1 + e^{-s}}$$

Many others: $tanh$, rectifier, etc.

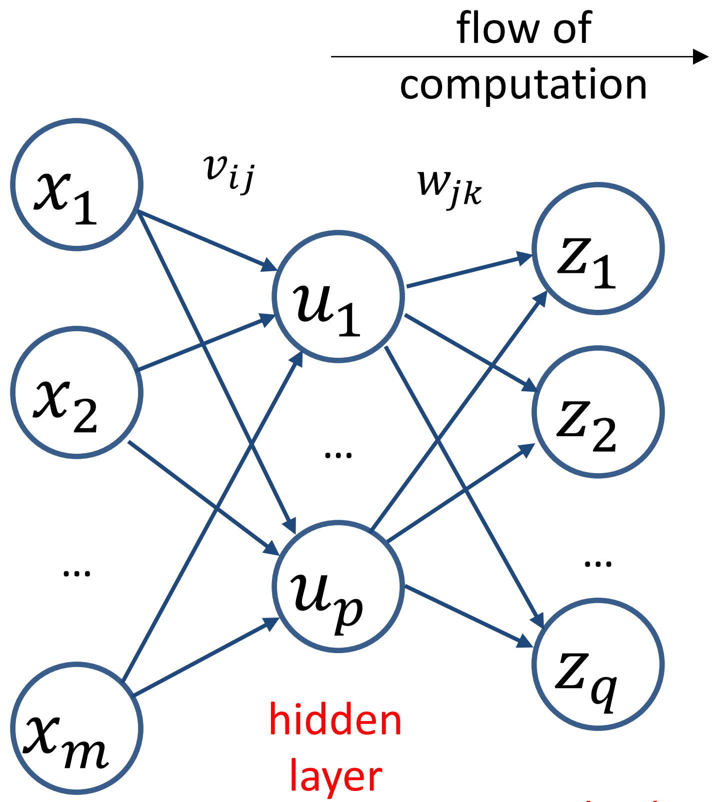


Feed-forward Artificial Neural Network

x_i are inputs, i.e., attributes

note: here x_i are components of a single training instance $\boldsymbol{x}$

a training dataset is a set of instances



input layer

hidden layer

output layer

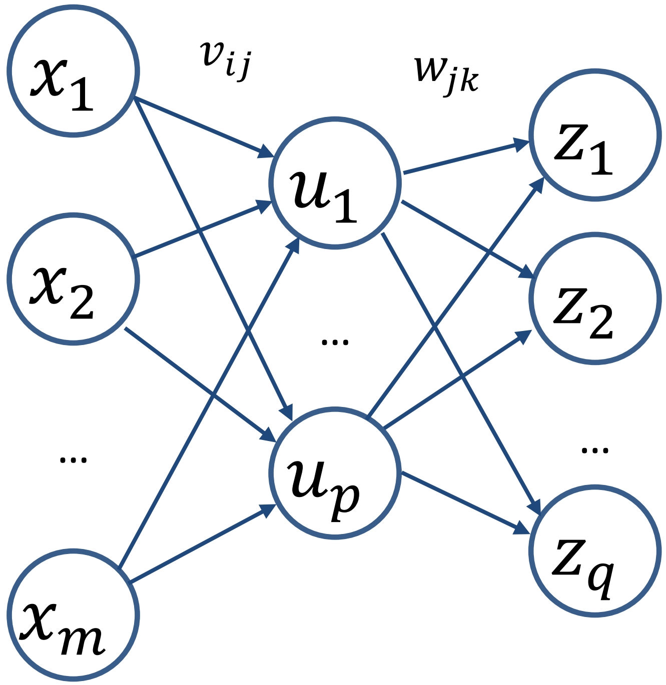
(ANNs can have more than one hidden layer)

z_i are outputs, i.e., predicted labels

note: ANNs naturally handle multidimensional output

e.g., for handwritten digits recognition, each output node can represent the probability of a digit

ANN as function composition



$$u_j = g(r_j)$$

$$r_j = v_{0j} + \sum_{i=1}^{m} x_i v_{ij}$$

$$z_k = h(s_k)$$

$$s_k = w_{0k} + \sum_{j=1}^{p} u_j w_{jk}$$

note that z_k is a function composition (a function applied to the result of another function, etc.)

here g, h are activation functions. These can be either same (e.g., both sigmoid) or different

you can add bias node $x_0 = 1$ to simplify equations: $r_j = \sum_{i=0}^{m} x_i v_{ij}$

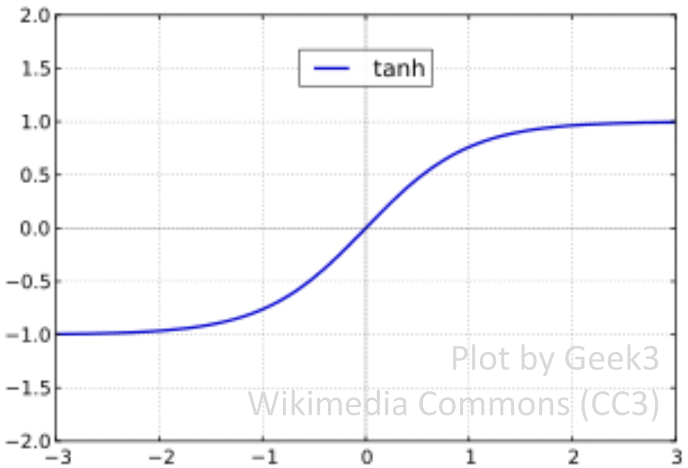
similarly you can add bias node $u_0 = 1$ to simplify equations: $s_k = \sum_{j=0}^{p} u_j w_{jk}$

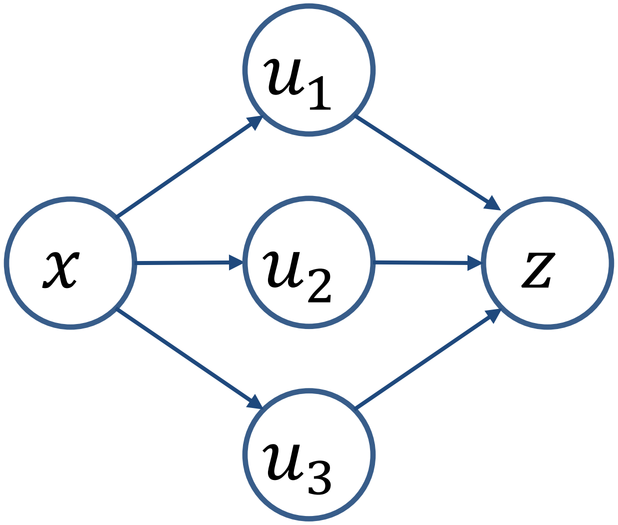
ANN in supervised learning

- ANNs can be naturally adapted to various supervised learning setups, such as univariate and multivariate regression, as well as binary and multilabel classification
- Univariate regression $y = f(\boldsymbol{x})$
  - e.g., linear regression earlier in the course
- Multivariate regression $\boldsymbol{y} = f(\boldsymbol{x})$
  - predicting values for multiple continuous outcomes
- Binary classification
  - e.g., predict whether a patient has type II diabetes
- Multiclass classification
  - e.g., handwritten digits recognition with labels “1”, “2”, etc.

The power of ANN as a non-linear model

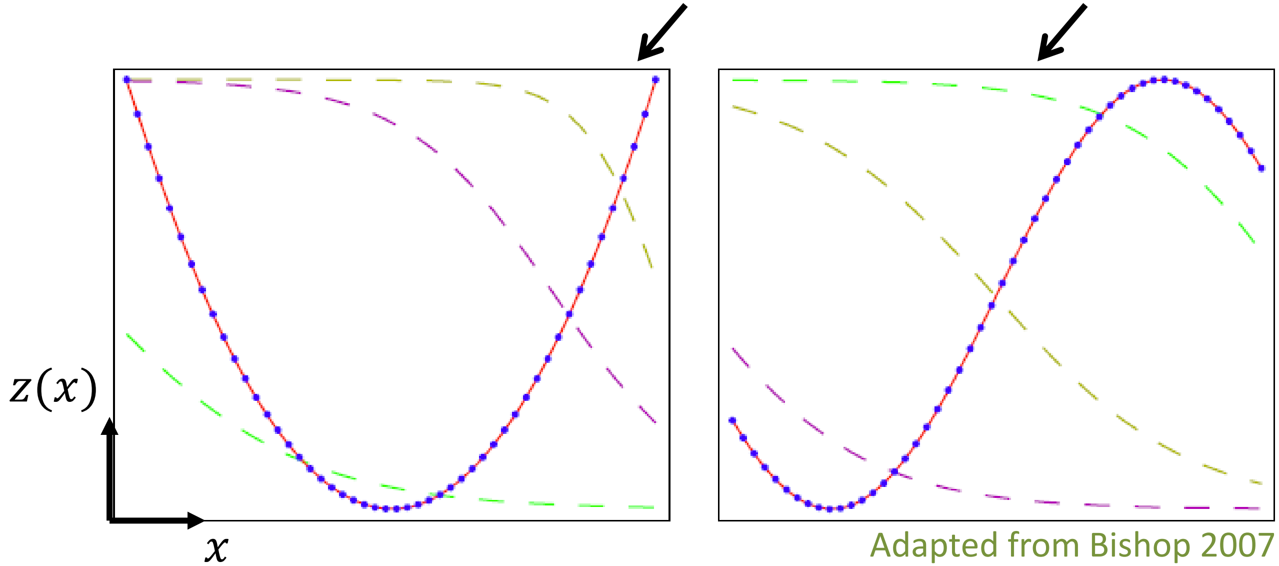
- ANNs are capable of approximating plethora non-linear functions, e.g., $z(x) = x^2$ and $z(x) = \sin x$
- For example, consider the following network. In this example, hidden unit activation functions are tanh





The power of ANN as a non-linear model

- ANNs are capable of approximating various non-linear functions, e.g., $z(x) = x^2$ and $z(x) = \sin x$



Adapted from Bishop 2007

Blue points are the function values evaluated at different x. Red lines are the predictions from the ANN. Dashed lines are outputs of the hidden units

- *Universal approximation theorem* *(Cybenko 1989)*: An ANN with a hidden layer with a finite number of units, and mild assumptions on the activation function, can approximate continuous functions on compact subsets of $\boldsymbol{R}^n$ arbitrarily well

Mini Summary

- Multiple layer networks
  - Model structure
  - Universal approximation

Training ANNs

Using iterative SGD technique

How to train your ~~dragon~~ network?

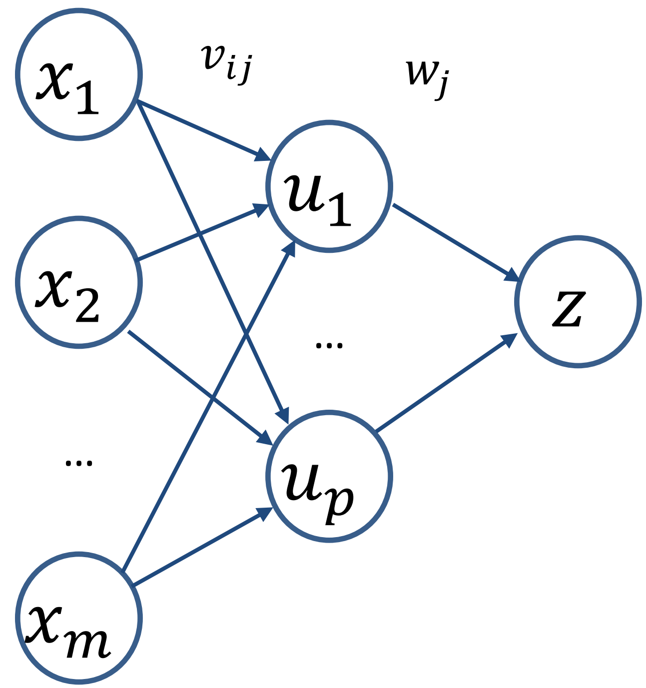
- You know the drill: Define the loss function and find parameters that minimise the loss on training data

Adapted from Movie Poster from Flickr user jdxyw (CC BY-SA 2.0)

- In the following, we are going to use stochastic gradient descent with a batch size of one. That is, we will process training examples one by one

Training setup: univariate regression

- Consider regression
- Moreover, we'll use identity output activation function $z = h(s) = s = \sum_{j=0}^{p} u_j w_j$
- This will simplify description of backpropagation. In other settings, the training procedure is similar



Loss function for ANN training

- Need loss between training example $\{\boldsymbol{x}, y\}$ & prediction $\hat{f}(\boldsymbol{x}, \boldsymbol{\theta}) = z$, where $\boldsymbol{\theta}$ is parameter vector of v_{ij} and w_j
- As regression, can use squared error

$$L = \frac{1}{2}\left(\hat{f}(\boldsymbol{x}, \boldsymbol{\theta}) - y\right)^2 = \frac{1}{2}(z - y)^2$$

(the constant is used for mathematical convenience, see later)

- Decision-theoretic training: minimise L w.r.t $\boldsymbol{\theta}$
  * Fortunately $L(\boldsymbol{\theta})$ is differentiable
  * Unfortunately no analytic solution in general

Stochastic gradient descent for ANN

Choose initial guess $\boldsymbol{\theta}^{(0)}, k = 0$

Here $\boldsymbol{\theta}$ is a set of all weights form all layers

For i from 1 to T (epochs)

For j from 1 to N (training examples)

Consider example $\{\boldsymbol{x}_j, y_j\}$

Update: $\boldsymbol{\theta}^{(\boldsymbol{k}+1)} = \boldsymbol{\theta}^{(\boldsymbol{k})} - \eta \nabla L(\boldsymbol{\theta}^{(\boldsymbol{k})})$; $k \leftarrow k+1$

$$L = \frac{1}{2}\left(z_j - y_j\right)^2$$

Need to compute partial derivatives $\frac{\partial L}{\partial v_{ij}}$ and $\frac{\partial L}{\partial w_j}$

Summary

- Perceptron
  - Introduction to Artificial Neural Networks
  - The perceptron model
  - Training algorithm
- Multiple layer networks
  - Model structure
  - Universal approximation
  - Training preliminaries
- Next lecture: Backprop training