# BCF mini course: Deep Learning and Macro-Finance Models

Goutham Gopalakrishna

École Polytechnique Fédérale de Lausanne (EPFL)

Swiss Finance Institute (SFI)

VSRC, Princeton University

February, 2023

Princeton University

# Target audience

- Econ/ORFE grad students and researchers interested in solving macro-finance models to study the global dynamics of an economic system
- Pre-requesites
    1. Basic numerical methods (Newton method, Finite differences etc.)
    2. ECO529 (Princeton) or equivalent
    3. Familiarity with any programming language, preferably Python 3.x or MATLAB.
    4. Good to have some familiarity with Objected Oriented Programming principles and Tensorflow 2.x

# Agenda

- Part-1: Introduction to neural networks
    - Why neural networks and deep learning
    - Function approximators
    - Comparison with existing methods
- Part-2: Deep learning principles, high-dimensional optimization techniques in machine learning
    - Gradient descent and variants
    - Under the hood: Activation functions, Parameter initialization
    - Object oriented programming principles
- Part-3: Application to solve macro-finance models with aggregate shocks

# References

- Textbooks:
  1. Raul Rojas. Neural Networks: A Systematic Introduction. 1996
  2. Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. An MIT Press book. 2016
- Other sources
  1. Dive into deep learning (interactive learning material)
  2. Machine learning for macroeconomics (teaching slides) by Jesús Fernández-Villaverde
  3. Neural networks (teaching slides) by Hugo Larochelle
  4. Deep learning CS6910 (teaching slides) by Mitesh Khapra

# Agenda

- Part-1: Introduction to numerical methods, challenges faced by traditional methods
  - Why neural networks and deep learning
  - Function approximators
  - Comparison with existing methods
- Part-2: Deep learning principles, high-dimensional optimization techniques in machine learning
  - Gradient descent and variants
  - Under the hood: Activation functions, Parameter initialization
  - Object oriented programming principles
- Part-3: Application to solve macro-finance models with aggregate shocks

# Introduction

- The basic idea of machine learning goes back to Rosenblatt (1958) who introduced the idea of perceptron
- The progress halted during the 1990s
- Forces behind the revival
  - Big data
  - Cheap computational power
  - Advancements in algorithms
- Popularity in industry: packages in Python, Tensorflow, Pytorch etc.
- Strong community support for packages $\implies$ better tools in the future
- Coding and compiling deep learning algorithms is easy thanks to the rich ecosystem provided by Pytorch, Tensorflow, Keras etc.

# Deep learning introduction

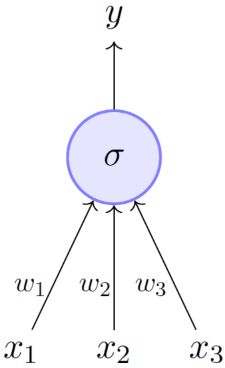
- The goal is to approximate a function $y = f(\boldsymbol{x})$, where $y$ is some scalar and $\boldsymbol{x}$ is a vector of inputs
- In basic econometrics, this is a regression problem. In macroeconomics, $f$ can be a value function, policy function, pricing kernel etc.
- $y$ can also be a vector (vector of value functions, probability distribution etc.)

# Deep learning introduction

- An artifical neural network (ANN) as an approximation to the function $f(\boldsymbol{x})$ takes the form

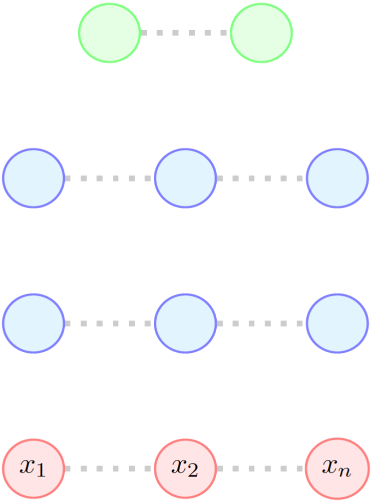
$$y = f(\boldsymbol{x}) \approx \sigma\left(\sum_{i=1}^{L} w_i x_i\right)$$

- The most fundamental unit of deep neural network is called an artificial neuron



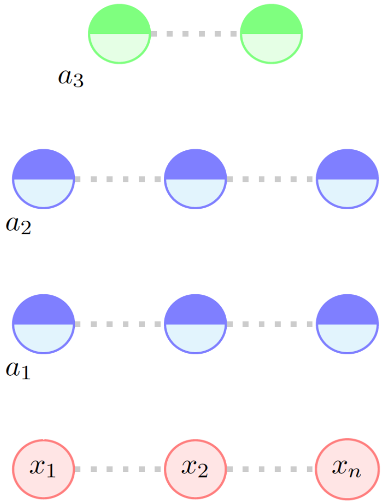
Artificial Neuron

# Feed forward neural network



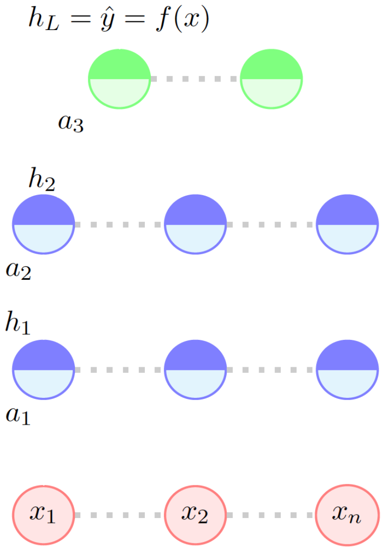
- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layers is called $0^{th}$ layer and the output layer is $L^{th}$ layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation ($a$) and activation ($h$)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n\times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n\times k}$ and $b_L \in \mathbb{R}^k$

# Feed forward neural network



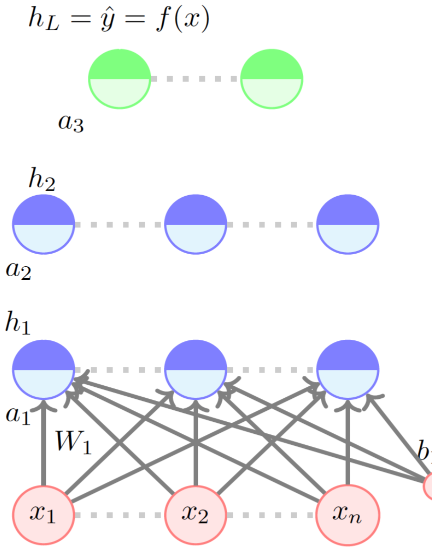
- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layers is called $0^{th}$ layer and the output layer is $L^{th}$ layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation ($a$) and activation ($h$)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n\times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n\times k}$ and $b_L \in \mathbb{R}^k$

# Feed forward neural network



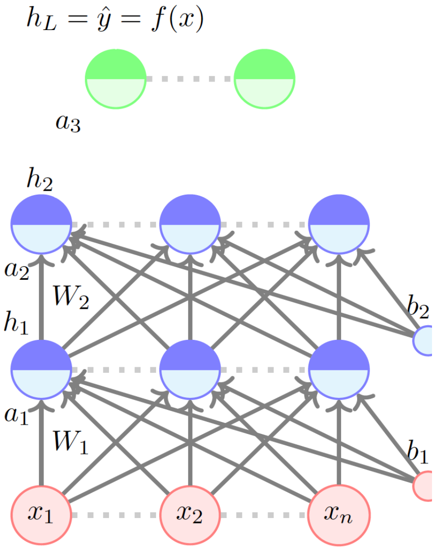
- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layers is called $0^{th}$ layer and the output layer is $L^{th}$ layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation ($a$) and activation ($h$)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n \times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$

# Feed forward neural network



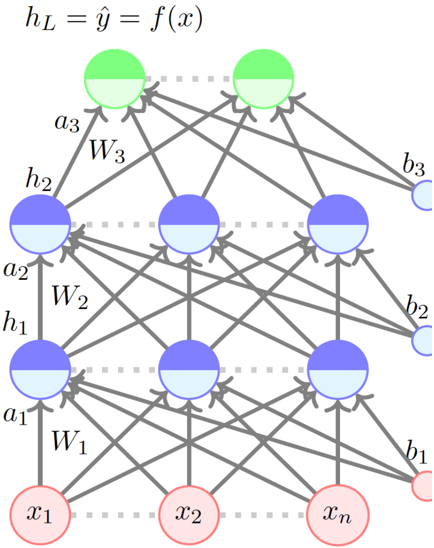
- The input is an n-dimensional vector
- The network contains $L - 1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layers is called $0^{th}$ layer and the output layer is $L^{th}$ layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation ($a$) and activation ($h$)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n \times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$

# Feed forward neural network



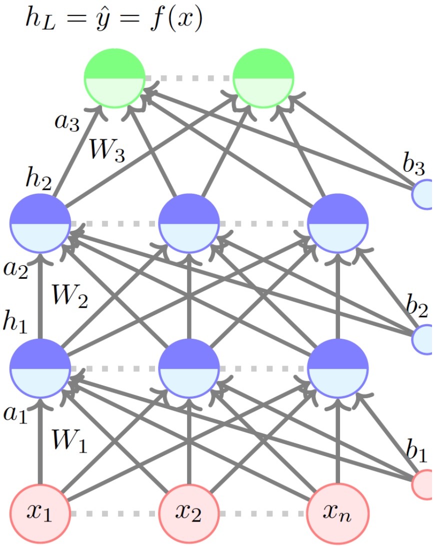
- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layers is called $0^{th}$ layer and the output layer is $L^{th}$ layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation ($a$) and activation ($h$)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n \times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$

# Feed forward neural network



- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having ***n*** neurons
- The input layers is called $0^{th}$ layer and the output layer is $L^{th}$ layer
- Finally, there is one output layer containing ***k*** neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation ($a$) and activation ($h$)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n \times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$

# Feed forward neural network: Mathematical representation



- The aggregation in layer $i$ is given by

$$a_i(x) = b_i + W_i h_{i-1}(x)$$

- The activation in layer $i$ is given by

$$h_i(x) = \sigma(a_i(x))$$

  where $g$ is called as the activation function

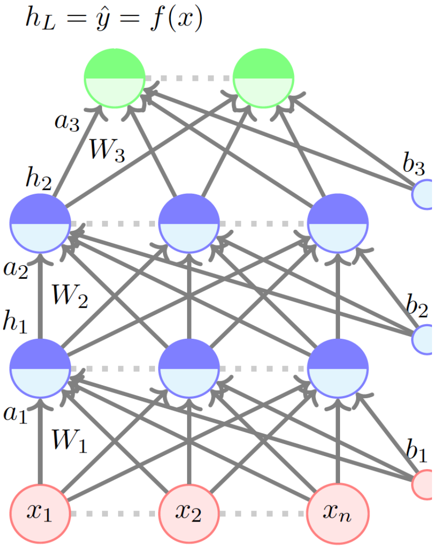
- The activation at the final layer is given by

$$\hat{y}(x) = O(a_L(x))$$

  where $O$ is the activation function on the final layer

- For simplicity, we will denote $a_i$ and $h_i$

# Feed forward neural network: Mathematical representation



- The aggregation in layer *i* is given by

$$a_i = b_i + W_i h_{i-1}$$

- The activation in layer *i* is given by

$$h_i = \sigma(a_i)$$

where *g* is called as the activation function on hidden layers

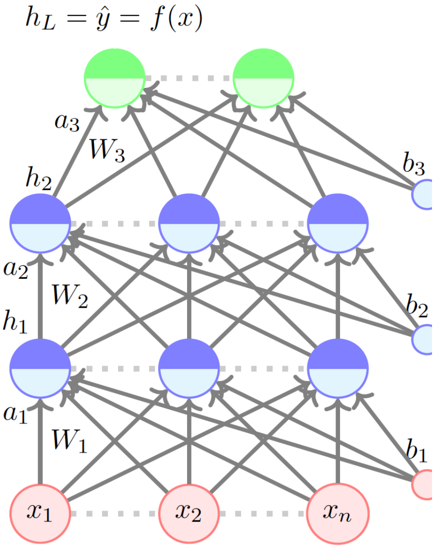
- The activation at the final layer is given by

$$\hat{y} = O(a_L)$$

where *O* is the activation function on the final layer

- For simplicity, we will denote $a_i$ and $h_i$

# Typical problem



- Data: $\{x_i, y_i\}$
- Model:

$$\begin{aligned}\hat{y}_i &= f^{DNN}(x_i)\\ &= O\Big(W_3\sigma(W2\sigma(W_1x + b_1) + b_2) + b_3\Big)\end{aligned}$$

- The type of neural network, number of layers, number of neurons in each layer, and activation function consistitute architecture of a particular neural network
- Parameters: $\theta = (W_1, ..., W_L; b_1, ..., b_L)$ where $L = 3$
- Goal is to learn the optimal parameters $\theta$ using an efficient algorithm

# Why deep learning works?

1. Finds representations of data that is informationally efficient
2. Convenient representation of geometry in high-dimensional manifold
    - Deep neural networks are chains of affine transformations- makes affine transformation followed by non-linear transformations sequentially
    - The chains of affine transformations ends up transforming the geometry of the state space
    - Optimizing in transformed geometry is often simpler

# Why deep learning works?

- Deep neural network is represented mathematically as

$$\hat{y} = f^{DNN}(\boldsymbol{x}) = O\big(W_3\sigma(W2\sigma(W_1\boldsymbol{x} + b_1) + b_2) + b_3\big)$$

  where the parameter vector is $\theta = (W_1, ..., W_L; b_1, .., b_L)$ and $O$ and $\sigma$ are activation functions

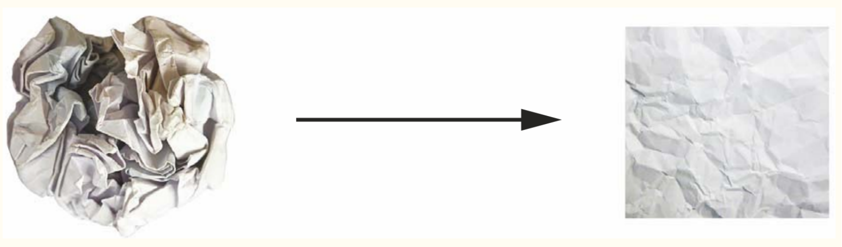
- Comparing this with a standard projection method

$$\hat{y} = f^{Proj}(\boldsymbol{x}) = \sum_{i=1}^{L} b_i\phi_i(\boldsymbol{x})$$

  where the parameter vector is $(b_1.,, b_L)$ and $\phi_i$ is a Chebychev polynomial

- Deep neural networks contain lots of parameters but with simple basis functions. Why is this useful? Because the sequence of affine and non-linear transformations ends up changing the geometry of the state space
- Finding convenient geometric representations of the data is more important than finding the right basis functions for approximation problems. This is where deep learning shines!

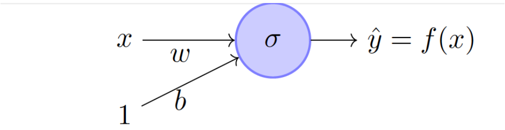
# Geometric transformation



Source: Jesus Fernandez-Villaverde

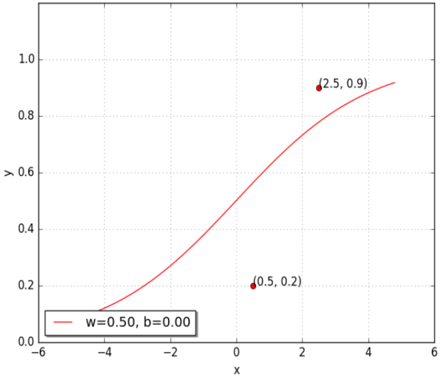
# Typical problem

- The problem at hand is to find the approximation $\hat{y} = f^{ANN}(\boldsymbol{x};\theta)$
- Assume that $f^{ANN}$ is a simple single layer network with activation $\sigma(\cdot) = \frac{1}{exp(-(wx+b))}$
- Consider a simple one dimensional problem. That is, the goal is to fit $(x,y) = (0.5, 0.2)$ and $(x,y) = (2.5, 0.9)$
- That is, the at the end of training the network, we would like to find $\theta*$ such that $f^{ANN}(0.5) = 0.2$ and $f^{ANN}(2.5) = 0.9$
- The parameter vector $\theta = [w,b]$ contain the weight and bias of the neuron activated $\sigma$
- The loss function is given by $\mathcal{L}(w,b) = \sum_{i=1}^{2}\left(y_i - f^{ANN}(x_i)\right)$



$$f(x) = \frac{1}{1+e^{-(w\cdot x+b)}}$$

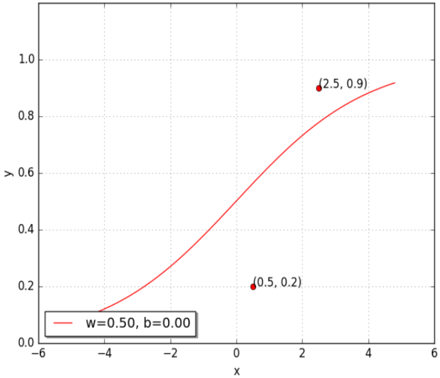
# Learning by trial and error



- Can we try to find $w*$, $b*$ manually?

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

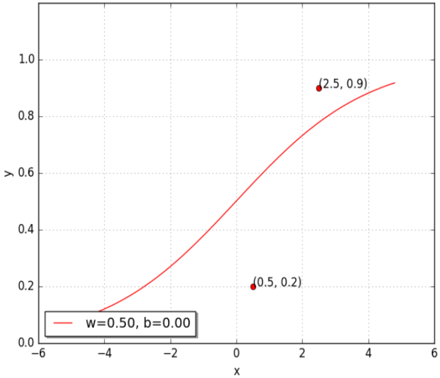
# Learning by trial and error



- Can we try to find $w*, b*$ manually?
- Let us use a random guess $(w = 0.5, b = 0)$

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

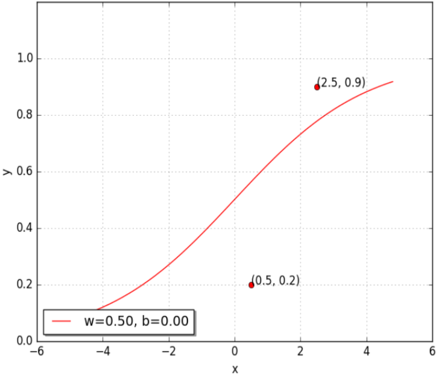
# Learning by trial and error



- Can we try to find $w*, b*$ manually?
- Let us use a random guess ($w = 0.5, b = 0$)
- Does not seem a great fit. How can we quantify how terrible ($w = 0.5, b = 0$) is?

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

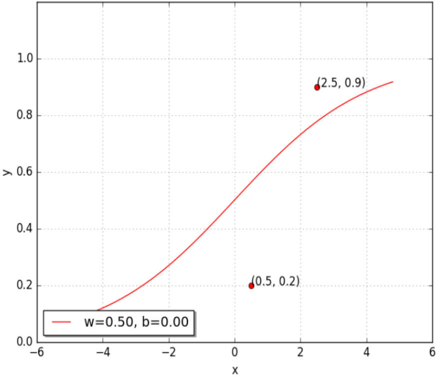
# Learning by trial and error



- Can we try to find $w*, b*$ manually?
- Let us use a random guess $(w = 0.5, b = 0)$
- Does not seem a great fit. How can we quantify how terrible $(w = 0.5, b = 0)$ is?
- Compute the loss using the loss function $\mathcal{L}(w, b) = \sum_{i=1}^{2} \left(y_i - f^{ANN}(x_i)\right)$

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

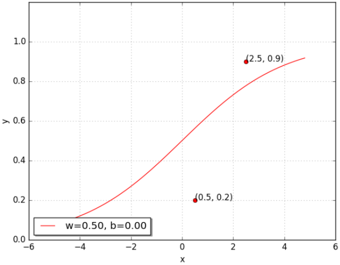
# Learning by trial and error



- Can we try to find $w*, b*$ manually?
- Let us use a random guess $(w = 0.5, b = 0)$
- Does not seem a great fit. How can we quantify how terrible $(w = 0.5, b = 0)$ is?
- Compute the loss using the loss function $\mathcal{L}(w, b) = \sum_{i=1}^{2} \left(y_i - f^{ANN}(x_i)\right)$
- $\mathcal{L}(0.5, 0) = 0.073$
- The goal is to make $\mathcal{L}(w, b)$ as close to zero as possible

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

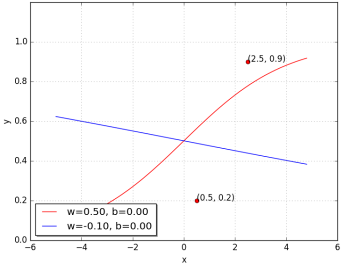
# Learning by trial and error



Let us try some other values of w, b

| $w$ | $b$ | $\mathscr{L}(w,b)$ |
|---|---|---|
| 0.50 | 0.00 | 0.0730 |

# Learning by trial and error

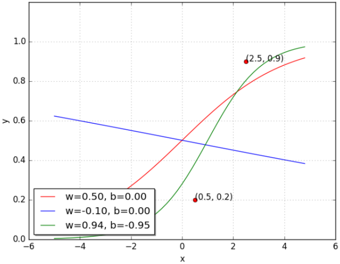


Let us try some other values of w, b

| $w$ | $b$ | $\mathscr{L}(w,b)$ |
|---|---|---|
| 0.50 | 0.00 | 0.0730 |
| -0.10 | 0.00 | 0.1481 |

It has made things worse. Perhaps it would help to push *w* and *b* in the other direction.

# Learning by trial and error

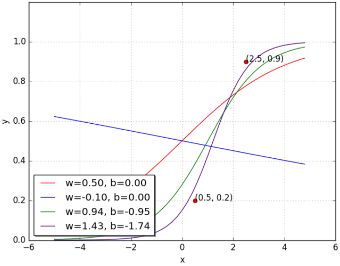


Let us try some other values of w, b

| $w$ | $b$ | $\mathscr{L}(w,b)$ |
|---|---|---|
| 0.50 | 0.00 | 0.0730 |
| -0.10 | 0.00 | 0.1481 |
| 0.94 | -0.94 | 0.0214 |

Much better. Let us keep going in this direction (i.e., increase $w$ and decrease $b$)

# Learning by trial and error

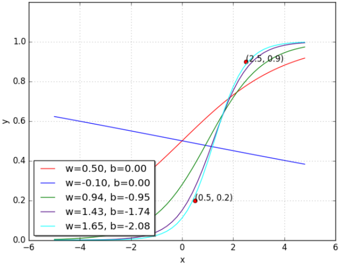


Let us try some other values of w, b

| $w$ | $b$ | $\mathscr{L}(w,b)$ |
|---|---|---|
| 0.50 | 0.00 | 0.0730 |
| -0.10 | 0.00 | 0.1481 |
| 0.94 | -0.94 | 0.0214 |
| 1.42 | -1.73 | 0.0028 |

Much better. Let us keep going in this direction (i.e., increase $w$ and decrease $b$)

# Learning by trial and error

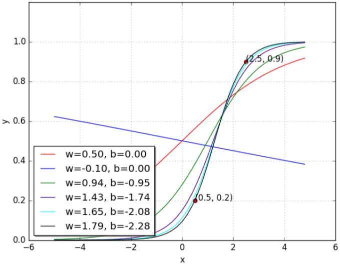


Let us try some other values of w, b

| $w$ | $b$ | $\mathscr{L}(w,b)$ |
|---|---|---|
| 0.50 | 0.00 | 0.0730 |
| -0.10 | 0.00 | 0.1481 |
| 0.94 | -0.94 | 0.0214 |
| 1.42 | -1.73 | 0.0028 |
| 1.65 | -2.08 | 0.0003 |

Much better. Let us keep going in this direction (i.e., increase $w$ and decrease $b$)

# Learning by trial and error

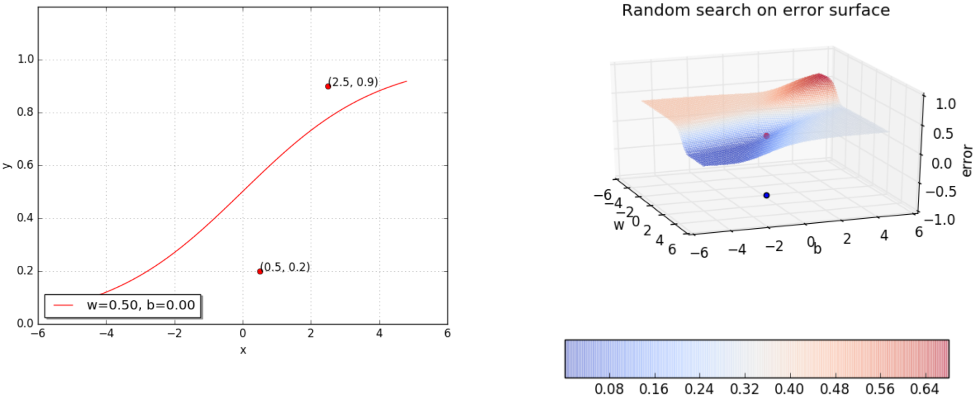


Let us try some other values of w, b

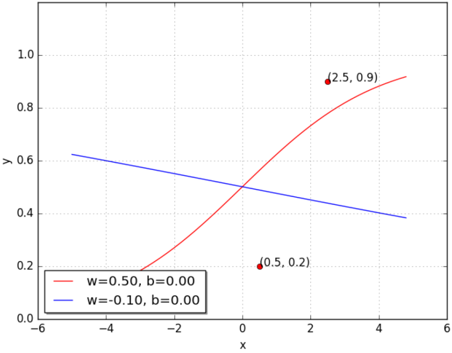
| $w$ | $b$ | $\mathscr{L}(w, b)$ |
|---|---|---|
| 0.50 | 0.00 | 0.0730 |
| -0.10 | 0.00 | 0.1481 |
| 0.94 | -0.94 | 0.0214 |
| 1.42 | -1.73 | 0.0028 |
| 1.65 | -2.08 | 0.0003 |
| 1.78 | -2.27 | 0.0000 |

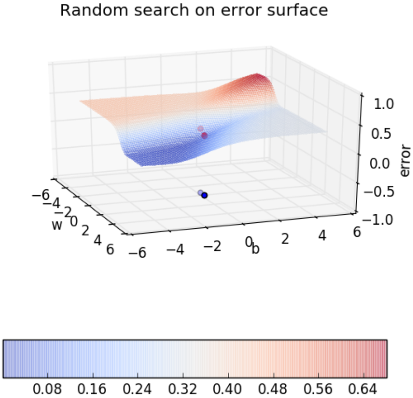
More principled way of doing this guesswork is what learning is all about!

# Learning by trial and error

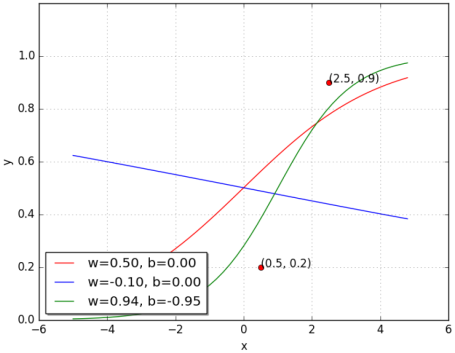


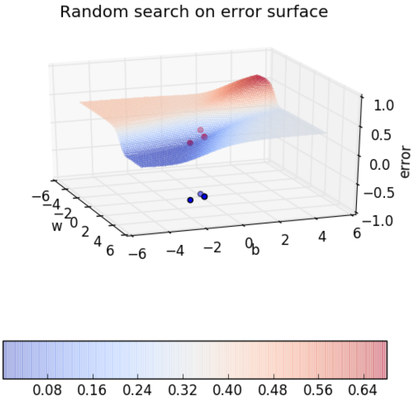
# Learning by trial and error



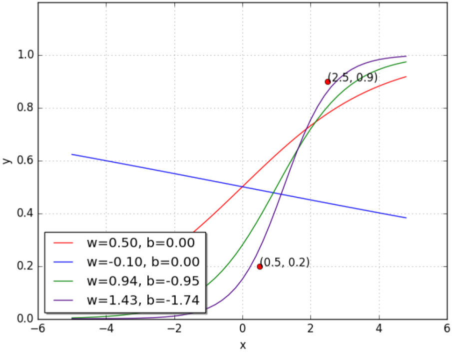


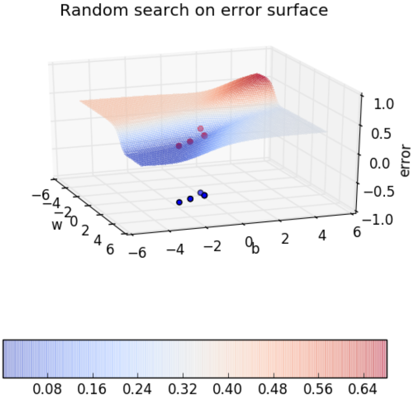
# Learning by trial and error



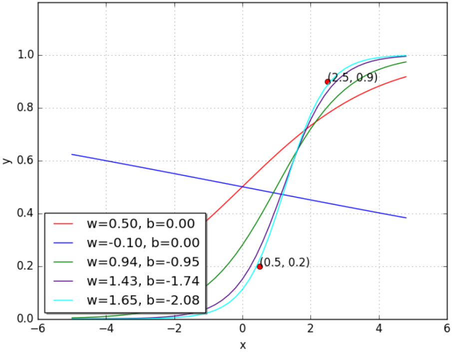

Random search on error surface

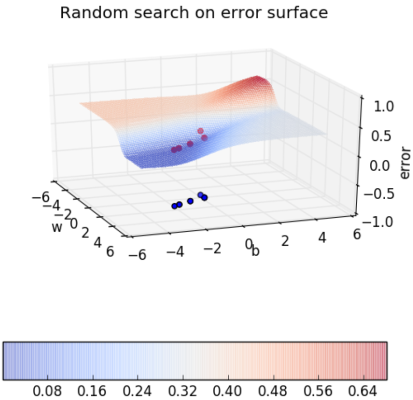
# Learning by trial and error



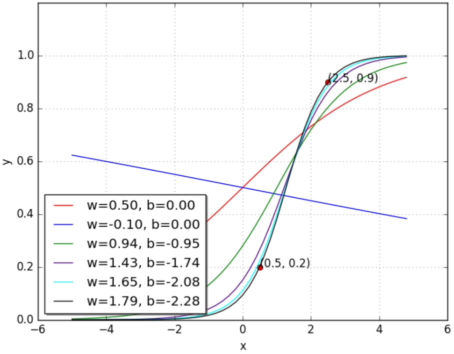


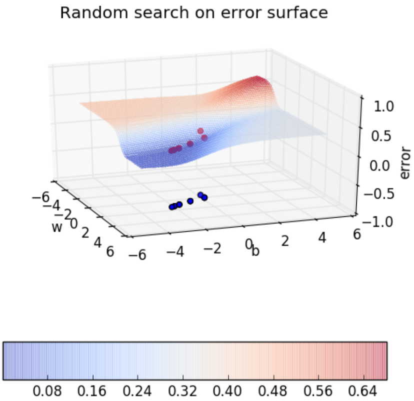
# Learning by trial and error




Random search on error surface

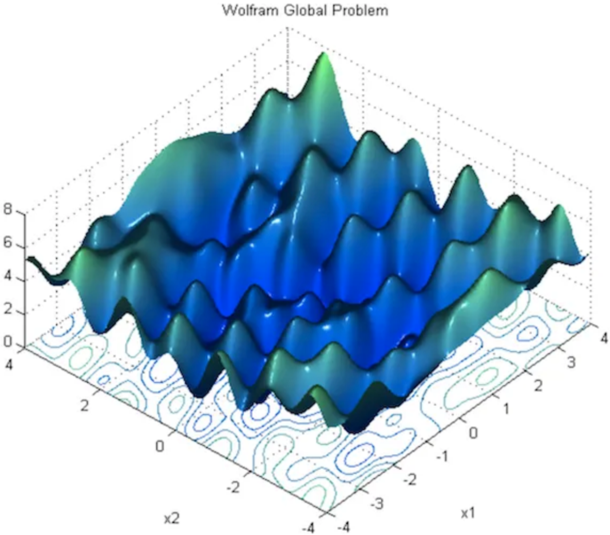
# Learning by trial and error





# Why deep neural networks?

- It seems like a single layer is enough to approximate the function well. Why do we need hidden layers?
- Complex problems require deep neural networks



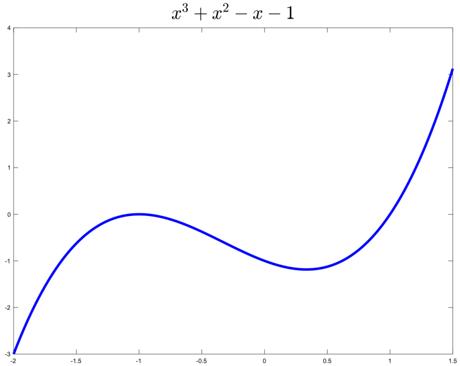
# Functional approximation

- **Universal approximation theorem** (Hornik, Stinchcombe, and White (1989)): A neural network with at least one hidden layer can approximate any Borel measureable function to any degree of accuracy
- However, having non-linear activation function in the hidden layers is important
    - Question: what happens when the activation functions are linear in a deep neural network?
- Once activation function is $\sigma(x) = \frac{1}{1+exp(-(wx+b))}$
- Another popular activation function is the Rectified Linear Unit (ReLU) $\sigma(x) = \max\{0, x\}$

# Function approximation example

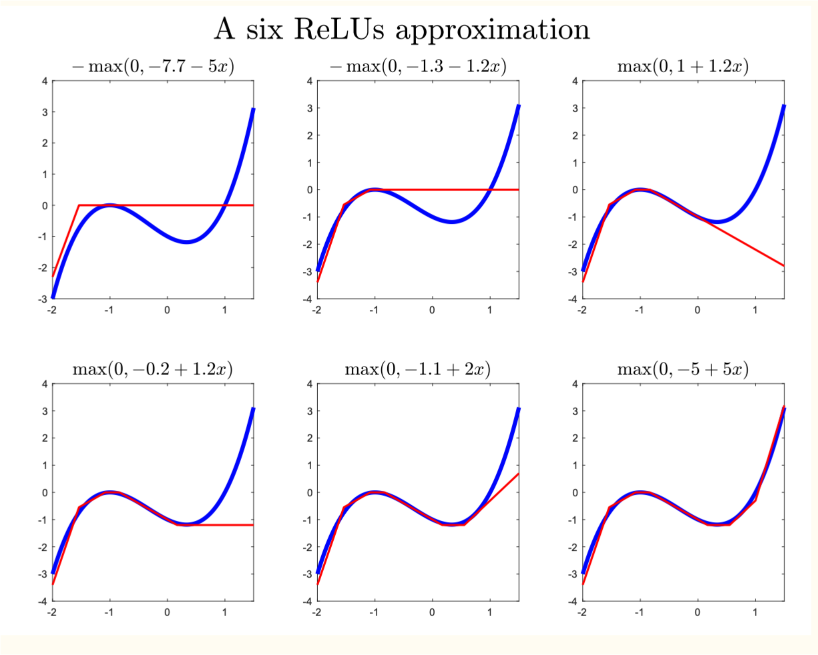
Let's try to approximate a one-dimensional function $f(x) = x^3 + x^2 - x - 1$ using a deep neural network with the following architecture

- Feed-forward neural network
- Six layers with one neuron in each layer
- ReLU activation function



Source: Jesus Fernandez-Villaverde

# Function approximation example



Source: Jesus Fernandez-Villaverde

# Comparison to other methods

Note that other methods can also approximate $f(x) = x^3 + x^2 - x - 1$ well but DNNs

- can also approximate functions with discontinuities. No assumptions about continuity or differentiability required
- can approximate high dimensional functions with better accuracy

| | High dimensions | Non-convex state space | Big data | Discontinuous functions | Global dynamics |
|---|---|---|---|---|---|
| Projection method | ✓ | ✗ | ✓ | ✗ | ✓ |
| Gaussian processes | ✓ | ✓ | ✗ | ✗ | ✓ |
| Adaptive sparse grid | ✓ | ✗ | ✓ | ✓ | ✓ |
| Deep learning: simulation | ✓ | ✓ | ✓ | ✓ | ✗ |
| Deep learning: active learning | ✓ | ✓ | ✓ | ✓ | ✓ |

# Limitations

Obviously, there are some limitations

- Deep neural networks require lots of data to work with
  - ➢ Not a problem for the task at our hand since we will use simulated data
- No theoretical guidance for choosing the right architecture
- Learning can be slow without access to a high performance cluster

# Software

- Install Python 3.x
- Install Tensorflow 2.x and Keras latest version
- Open a google colab account (free)
- Access to high performance computing cluster?