# An Introduction to Solving Macroeconomic Models with Neural Networks

Raphaël Huleux [1]

[1]University of Copenhagen

Banque de France — 29 January 2026

# Why Deep Learning in Macroeconomics?

- Solving most models implies a **dynamic maximization problem**
- **Dynamic programming** is subject to the **curse of dimensionality**
  $\rightarrow$ Number of computation grows exponentially with number of states
- We can't solve many interesting problems!
  $\rightarrow$ e.g. household problem with safe, risky, mortgage and housing?
- In GE, with HA, problem becomes even more untractable

# The solution is DL?

Main idea in a nutshell:

- Use a **simulation method** instead of a grid
- Approximate value / policy functions with a **neural network**
- Improve the solution by using (stochastic) **gradient descent**
- Get derivatives using **auto-differentiation**

In practice:

- Use state-of-the-art libraries (Pytorch, Tensorflow, JAX, ...)
- Code is **simpler** than DP
- Bonus: **easy GPU** computing as well

# Drawbacks and Limits

- A lot of hyper-parameters (learning rate, size of networks, etc)
- Much slower than DP for small models
- Hard to measure accuracy for interesting models
- Compute can be expensive and long
- Young field: no benchmark / workhorse algorithms yet

# This talk

Outline:

1. Basics of Neural Networks
2. Solving the Ramsey model with DL
3. Adding uncertainty (= RBC model)
4. Adding constraints (= consumption-savings model)
5. An overview of the literature for solving HA + GE with DL

Important: Not my own research today!

Most of the results today loosely from Maliar, Maliar, Winant (2022)

# Basics of Neural Networks

# Definition of Neural Networks

A **neural network** is a parameterised function that maps an input vector $x$ to an output vector $\hat{y}$:

$$\hat{y} = f(x; \theta),$$

where $\theta$ denotes the collection of parameters (weights $W$ and biases $b$).
Each neuron does two things:

1. An affine transformation $z = \sum_i w_i x_i + b$;
2. Followed by a pointwise nonlinear transformation $a = \sigma(z)$,

where the output of one layer serves as the input to the subsequent layer.

# Why use neural networks?

Cybenko, 1989; Hornik et al. (1989)

> *A **feedforward neural network** with a single hidden layer containing a finite number of neurons **can approximate any continuous function on compact subsets of** $\mathbb{R}^n$ **to arbitrary precision**, provided the activation function satisfies certain mild regularity conditions.*

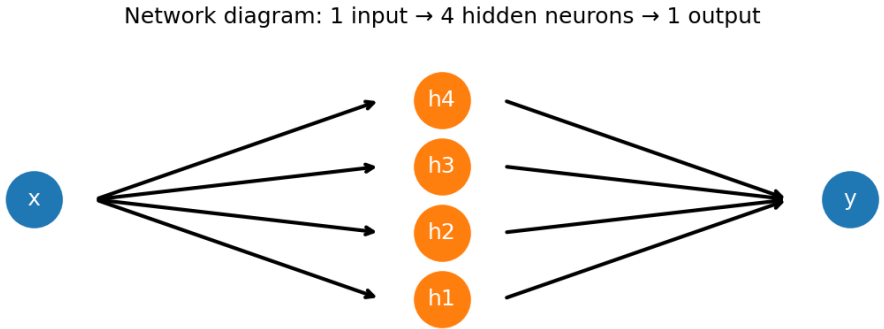
# Neural Network Notation

- A network takes as input a batch $x$ of size $(N^B, N^x)$
  - $N^B$ is the batch size (= number of "observations" in the dataset)
  - $N^x$ the number of input feature (= number of "variables)
- $a_j^{(\ell)}$ is the output of neuron $j$ in layer $\ell$, with $a_j^{(\ell)} = \sigma(z_j^{(\ell)})$
  - with $z_{b,j}^{(\ell)} = \sum_i^{N^{\ell-1,h}} a_{b,i}^{(\ell-1)} W_{i,j}^{(\ell)} + b_j^{(\ell)}$
  - and $\sigma(z_j^{(\ell)})$ a (nonlinear) activation function
- Stacking over all neurons in a layer, we can write the affine transformation as $z^{(\ell)} = a^{(\ell-1)} W^{(\ell)} + b^{(\ell)}$

# Today's example

- Train a network with one hidden layer to approximate the function $y = \sin(x)$
- Simplest setup: $N^x = N^y = 1$
- We can sample "for free" from the real function
- Implement it without any package! "homemade"

# Neural Network with One Layer



# Approximate a function with a network

Algorithm:

- Initialize parameters $\theta$ of the network
- Draw pairs of $(x, y)$ of the function we want to approximate (here, $y = \sin(x)$)
- For X iterations, do
  1. Using parameters $\theta$, predict $\hat{y} = f(x; \theta)$
  2. Evaluate the loss function $L(\theta) = \frac{1}{N^b} \sum_i (\hat{y}_i - y_i)^2$
  3. Compute gradients of loss function with respect to parameters
  4. Update parameters using gradient descent: $\theta' = \theta - \eta \nabla L(\theta)$

     $\rightarrow \eta$ = learning rate

Let's code!

# Solving the Ramsey Model with DL

# The Ramsey model

A household maximizes

$$V(K_{-1}) = \max_{\{K_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t u(C_t)$$

subject to

$$K_t + C_t = K_{t-1}^{\alpha} + (1-\delta)K_{t-1}.$$

The Euler equation writes

$$u'(C_t) = \beta(1 + \alpha K_t^{\alpha-1} - \delta)u'(C_{t+1})$$

# Solving with DL (Maliar, Maliar, Winant 2022)

Main idea

Approximate policy function using a neural network, targetting the Euler error.

Trick

instead of approximating $K_t = g(K_{t-1})$, use $s \equiv K_t/(f(K_{t-1}) + (1-\delta)K_{t-1}) \in (0,1)$

$\rightarrow$ all choices are feasible!

Other algorithms

In the paper, they show value function and lifetime reward approaches.

# Euler-residual minimization Algorithm

Algorithm:

- Initialize parameters $\theta$ of the network
- For X iterations:
  1. Draw a random sample of states $K_{t-1}$ over $(\underline{K}, \overline{K})$
  2. Using the current policy function, compute $C_t$ and $C_{t+1}$
  3. Evaluate the EE: $\beta u'(C_{t+1})(1+r)/u'(C_t) - 1$
  4. Compute the Mean Square Error $L(\theta) = \frac{1}{N}\sum_i EE_i^2$
  5. Update parameters using gradients

Let's code!

# Adding risk: the RBC model

# The RBC model

Now, assume TFP follows a log AR-1 process:

$$V(K_-) = \max_{\{K_t\}_{t=0}^{\infty}} \beta^t \mathbb{E}[u(C_t)]$$

subject to

$$K_t + C_t = Z_t K_{t-1}^{\alpha} + (1-\delta)K_t,$$

$$\log Z_t = \rho \log(Z_{t-1}) + \varepsilon_t$$

The Euler equation writes

$$u'(C_t) = \beta \mathbb{E}[(1 + \alpha Z_{t+1} K_t^{\alpha-1} - \delta) u'(C_{t+1})]$$

# Computing expectations

Main difficulty: How to compute the expectation?

Three options:

1. Use standard quadrature method (but subject to curse of dimensionality)
2. Use Monte-Carlo method
3. Use the two-in-one operator of Maliar, Maliar, Winant (2022)

# Adding borrowing constraints:

# The consumption-saving model