Using deep learning for dynamic programming

Jacob Røpke, Jeppe Druedahl

April 2024

Deep learning for solving dynamic models

- A fairly recent literature uses deep learning to solve models.
- Deep learning: The use of artificial neural networks (NNs from now on).
- This lecture:
    - Why use NNs?
    - How do we use NNs for solving models?

Plan

Why use neural networks?

Curse of dimensionality

- Dynamic programming (DP) face the curse of dimensionality: As the number of states grows time to solve grows exponentially.
  - Note: Number of states = Number of dimensions
- Key reason behind curse of dimensionality: Tensor grids grows exponentially as we add states.
- Attempts to solve this problem:
  - Choose grid points smartly to allow for less points
  - Use smarter interpolators to allow for less points.

How does NNs alleviate this problem

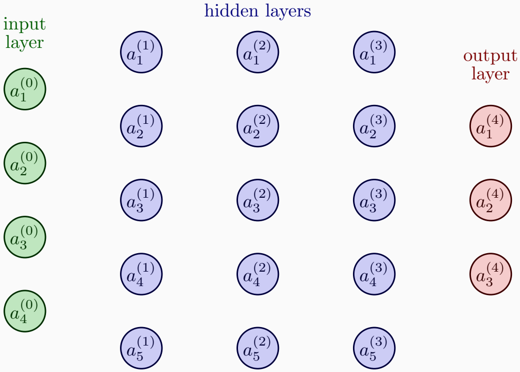
- NNs seem to be very effective at handling high-dimensional problems.
- Example: LLMs(think ChatGPT) handle text input. Text input is very high-dimensional.
- NNs are essentially effective interpolators
- Why are they effective?
  - We do not know

Problems

- Bertel: "Often no gaurantee of convergence"
- Typically slower in small models
- Can be sensitive to hyper-parameters
  - Hyper-parameter: Non-model parameter used for training. A DP-example could be the number of grid points.
- Hardware limitation: Access to a GPU is essential for speed.
- Black-box problem: We do not understand how the neural net works.

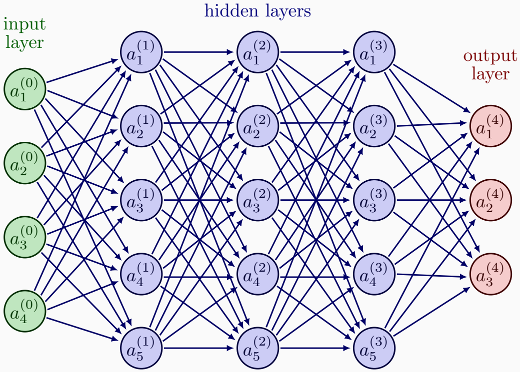
What are artifical neural networks?

Deep Neural networks



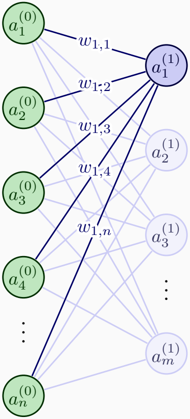
- **Neurons:** $a^{(layer)}_{\text{Neuron number}} \in \mathbb{R}$
- **Deep=** just means multiple hidden layers

Fully-connected feed-forward NNs



- **Feed Forward:** Unidirectional (left to right)
- **Fully Connected:** All neurons in neighboring layers connected

NNs: math



$$= \sigma\left(w_{1,1}a_1^{(0)} + w_{1,2}a_2^{(0)} + \ldots + w_{1,n}a_n^{(0)} + b_1^{(0)}\right)$$

$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}a_i^{(0)} + b_1^{(0)}\right)$$

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,n} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \ldots & w_{m,n} \end{pmatrix}\begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]$$

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{a}^{(0)} + \mathbf{b}^{(0)}\right)$$

- σ : **Activation function**
- w and b: **Weight and biases** (parameters, θ)

NN properties

- **Universal function approximation theorems**
  - For fixed number of layers: With enough neurons we can approximate any function.
- **Lots of parameters**

Model example

Model class

- Today: Continuous choice models.

Buffer-stock model

Today I will consider the case of a buffer-stock model.

$$
\begin{aligned}
\max_{c_0,\dots c_t,\dots,c_{T-1}} \sum_{t=0}^{T-1} \beta^t E_t[u(c_t)] \\
m_{t+1} = Ra_t + p_{t+1}\psi_{t+1} \\
a_t = m_t - c_t \geq 0 \\
p_{t+1} = p_t^{\rho_\xi}\xi_{t+1} \\
\psi_{t+1} \sim \log\mathcal{N}(-0.5\sigma_\psi^2, \sigma_\psi^2) \\
\xi_{t+1} \sim log\mathcal{N}(-0.5\sigma_\xi^2, \sigma_\xi^2)
\end{aligned}
$$

Neural network

We parametrize the policy function as a neural network with two inputs(one for each state) and one output - consumption:

$$c_t(m_t, p_t) = \pi(m_t, p_t, t; \theta)$$

where π is a feedforward neural network and θ are the parameters of that network.

- Problem: How to find θ to solve the buffer-stock model.
- How to include time as an input:
  - T different neural nets
  - Use t directly
  - Use dummies for each time period.

How to train a neural network

Training ingredients

Solving a model using a neural network requires finding the "optimal" set of parameters θ. We need two key ingredients:

1. A loss function to be minimized
2. Input data for minimizing (think state variables)

Problem: How do I update the very large set of parameters.

Parameter update: Gradient descent

Methods using second-order derivatives are infeasible when we have many parameters. Instead: (Almost) simple gradient descent:

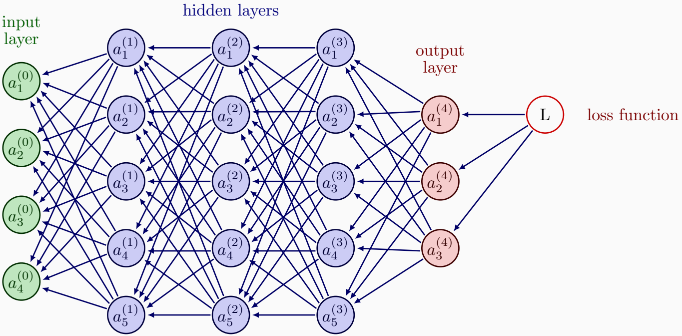
$$\theta^{i+1} = \theta^{i} - \alpha \nabla L(\theta^{i}; S)$$

where α is a learning rate, θ is the NN parameters, L is the loss-function and S is a sample of states.

- Problem: How to compute $\nabla L(\theta^{i}; S)$ for a given loss-function.

Backpropagation

- Tool: Automatic differentiation
- Idea: Use automatic differentiation to get derivative of loss-function wrt. to output of neural net. Then use chain rule backwards through the neural net.



Backpropagation

- We do not have to code this up ourselves: Use standard optimized packages - Tensorflow, Pytorch, Jax.

Solving equations

Equation-based methods

- Most of the literature on using deep learning for solving models originates in macroeconomics.
- Solving macro-models typically involve solving a system of equations.
- We can do the same to solve the buffer-stock model.

Equations to solve

- To solve the buffer-stock model, we need to solve the FOC (Euler)

$$u'(c_t) = \beta R E_t[u'(c_{t+1})]$$

- We also need to handle the borrowing-constraint.

$$a_t \geq 0$$

- Maliar, Maliar, Winant (2021) approach: Encode inequality constraint as an equality constraint using a Fischer-burmeister function.

Fischer-burmeister function

$$f(x, y) = \sqrt{x^2 + y^2} - (x + y) \tag{1}$$

Properties:

$$f(x, 0) \begin{cases} = 0 & \text{if } x \geq 0 \\ > 0 & \text{else} \end{cases} \tag{2}$$

$$f(0, y) \begin{cases} = 0 & \text{if } y \geq 0 \\ > 0 & \text{else} \end{cases} . \tag{3}$$

If we encode the inequality constraint as an equality condition, we can simply put it into the fischer-burmester along with the euler and then solve for the root of this function.

$$m_t - c_t = 0 \tag{4}$$

Loss-function

The loss function to be minimized is:

$$L(\theta; S) = \frac{1}{N(T-1)} \sum_{i=0}^{N-1} \sum_{t=0}^{T-2} f(\text{Euler}_{t,i}, \text{Borrowing constraint}_{t,i})^2 \quad (5)$$

We then need to specify input data S.

Note: We omit the terminal period (T-1) as the solution to consume everything can be found in closed form.

Input data

We need to feed the loss-function on which we can evaluate the equation error. Let us consider two options:

1. Using grids
2. Simulating data

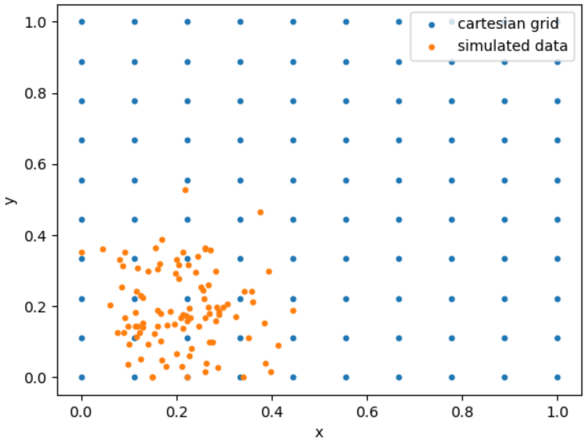
Using grids

- Problem: Grids are part of the reason why we face the curse of dimensionality in traditional DP.
- Advantage: A grid dataset is stable and easy to generate.
- Disadvantage: Cartesian grid grows exponentially as the state-space grows $\Rightarrow$ Curse of dimensionality.

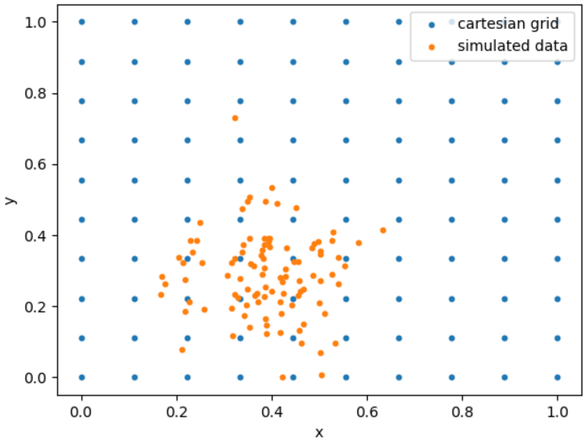
Simulating data

- We could also simulate data. For a given policy, function we just simulate N individuals for T periods to generate some data.
- We can then feed the state-sample from the simulation into the loss-function.
- Advantage: Does not grow exponentially in the number of states
  - We solve the model where the agents live.
- Disadvantage: Can be unstable

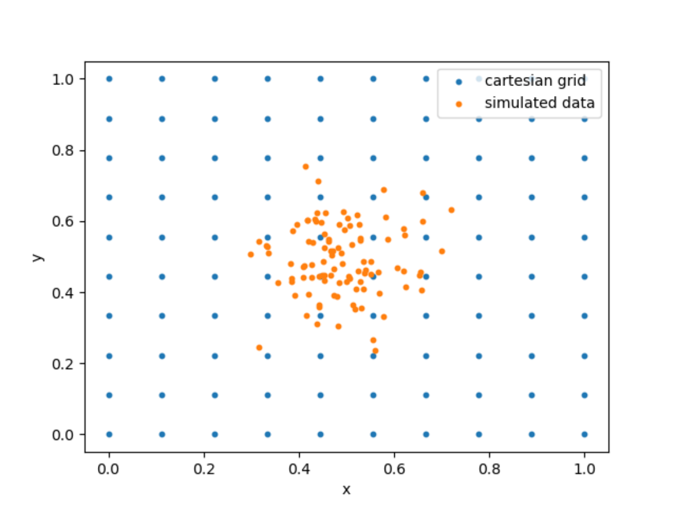
Why simulation can break the curse of dimensionality



Why simulation can break the curse of dimensionality



Why simulation can break the curse of dimensionality



Summarizing equation-based approach

Algorithmic steps:

1. Initialize policy function as a neural network
2. Simulate data with policy function
3. Compute loss-function by evaluating equations to compute the equation error across T and N in the in the simulated data.
4. use backpropagation to get gradients for all parameters
5. update parameters using gradients.
6. If equation error is not below some tolerance level: return to step 2.

Tensorflow code that solves a series of (macro) models using this approach

Azinovic code from DSE2023

Simulating utility

Buffer-stock model

Idea: Maximize expected discounted utility directly

$$
\begin{aligned}
\max_{c_t} \sum_{t=0}^{T-1} \beta^t E_t[u(c_t)] \\
m_{t+1} = Ra_t + p_{t+1}\psi_{t+1} \\
a_t = m_t - c_t \geq 0 \\
p_{t+1} = p_t^{\rho_\xi} \xi_{t+1} \\
\psi_{t+1} \sim \log \mathcal{N}(-0.5\sigma_\psi^2, \sigma_\psi^2) \\
\xi_{t+1} \sim log\mathcal{N}(-0.5\sigma_\xi^2, \sigma_\xi^2)
\end{aligned}
$$

Issues

- Problem 1: How to handle expectations?
  - Easiest solution: Monte carlo integration
  - Simulate N agents and take the average discounted tulity
- Problem 2 (only in infinite horizon)
  - Solution: Truncate the number of periods to T^{sim}
  - T^{sim} needs to be "large enough" to approximate an infinite horizon
  - Implication: Higher β requires higher T^{sim}

Loss function

The loss function is now:

$$L(\theta; S_0, \psi, \xi) = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{t=0}^{T-1} \beta^t u(c_t)$$

$$c_t(m_t, p_t) = \pi(m_t, c_t, t; \theta)$$

$$m_{t+1} = Ra_t + p_{t+1}\psi_{t+1}$$

$$a_t = m_t - c_t \geq 0$$

$$p_{t+1} = p_t^{\rho_\xi} \xi_{t+1}$$

Loss-function

- This only requires writing a simulation function: Very easy to implement!

Input data

The input data is initial states (S_0) and shocks (ψ, ξ) for the simulation. To avoid overfitting on a specific set of shocks and initial states, we will draw these randomly each iteration.

Termination

- It is less obvious when we should terminate the algorithm. An easy approach is just to terminate the algorithm when the loss stops improving.
- Problem: Randomness due to shocks and initial states affects the loss.
- Solution: Use a fixed set of states and shocks to evaluate whether the loss is improving.

Summarizing approach

1. Initialize policy function as a neural network
2. Draw initial states and shocks from their true distributions
3. Simulate the model forward and compute average discounted utility
4. use backpropagation to get gradients wrt. to all parameters θ
5. update parameters θ
6. Simulate again to get loss for fixed states and shocks. If no improvement for some chosen number of iterations, terminate the algorithm. Otherwise return to step 2.

Note: We may not want to simulate with fixed states/shocks every time as it significantly increases the computational cost of each iteration(at worst it is doubled). Instead we can simply do it with some pause between iterations.

Conclusion

- We can use neural networks to alleviate curse of dimensionality
- Disadvantages:
    - Stability
    - Precision
    - No guarantees of convergence
    - Requires GPUs to be effective