Deep Learning for Dynamic Programming

Jacob Røpke

April 2025

Motivation

Key problem in Dynamic Programming: Curse of Dimensionality

- Exponential growth of computational costs.
- This growth happens when adding:
  - More states
  - More choices
  - Also more shocks (Not the focus today).

Motivation

Key problem in Dynamic Programming: Curse of Dimensionality

- Exponential growth of computational costs.
- This growth happens when adding:
  - More states
  - More choices
  - Also more shocks (Not the focus today).
- How to alleviate this problem?

Motivation

Key problem in Dynamic Programming: Curse of Dimensionality

- Exponential growth of computational costs.
- This growth happens when adding:
  - More states
  - More choices
  - Also more shocks (Not the focus today).
- How to alleviate this problem?
  1. Model-specific reduction number of states,choices,shocks
  2. Better interpolators $\Longrightarrow$ Fewer grid-points
  3. Smart ways to pick grid-point $\Longrightarrow$ Fewer grid-points

Today: Focus on points 2 and 3.

Deep Learning for solving Economic Models

Recent literature: Deep learning can alleviate curse of dimensionality

Two tricks:

1. Instead of exogenously chosen grid-points: Use simulation to find grid-points
2. Use Neural networks for interpolation

Today

- Focus on **Continuous choice models**
- Lecture is based on joint work with Jeppe Druedahl.
- In the paper we compare multiple deep learning methods. I focus on the simplest today.

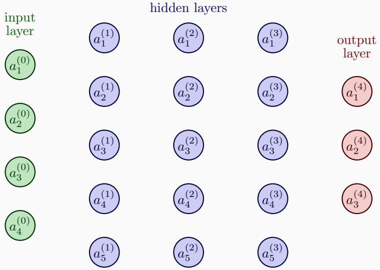
Today

- Focus on **Continuous choice models**
- Lecture is based on joint work with Jeppe Druedahl.
- In the paper we compare multiple deep learning methods. I focus on the simplest today.
- Plan:
    1. What is Deep learning
    2. Applying deep learning to solve a buffer-stock model
    3. Applying deep learning to solve a buffer-stock model with durable goods
    4. Code (If we get to it).

What is Deep Learning?

Deep Neural Network (DNN): Basic Structure

Neural Network: Network of interconnected "nodes".

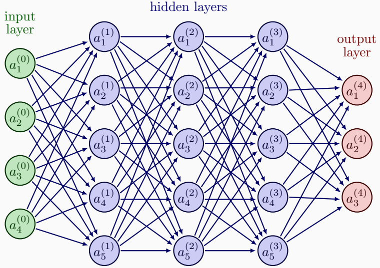


In economic models:

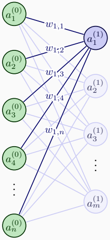
- Input nodes: **state variables**
- Output nodes: **choices**

DNN: How do nodes connect?

- This specific structure: Fully connected feedfoward network.
- Basic idea:
  1. Give vector of state variables as input
  2. Use state variables to compute first hidden layer.
  3. Use first hidden layer to compute second and so on.
  4. Finally compute choices from final hidden layer.



DNN: What are the arrows?



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

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

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

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

- Each arrow symbolises a weight: $w_{1,1}, w_{1,2}$.
- To compute one node in next layer $a_1^{(1)}$:
  1. Compute weighted sum of previous layer nodes and weights
  2. add constant $b_1^{(0)}$
  3. Finally put sum into some non-linear function σ.

Solving a Buffer-stock model

Buffer-stock model

Goal: Agents choose savings rate to maximize life-time utility.

Bellman equation for working agent:

$$v_t(m_t, p_t) = \max_{a_t} u(c_t) + \beta E_t[v_{t+1}(m_{t+1}, p_{t+1})]$$

$$\text{s.t}$$

$$c_t = (1 - a_t)m_t$$

$$m_{t+1} = (1 + r)(m_t - c_t) + \text{income}_{t+1}$$

$$\text{income}_{t+1} = \kappa_{t+1}\psi_{t+1}p_{t+1}$$

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

$$a_t \in [0; 1]$$

- ψ_{t+1}, ξ_{t+1} are log-normal shocks.
- Agents retire at age $T^{retired}$ after which they receive low deterministic income.

Buffer-stock model

Goal: Agents choose savings rate to maximize life-time utility.

Bellman equation for working agent:

$$v_t(m_t, p_t) = \max_{a_t} u(c_t) + \beta E_t[v_{t+1}(m_{t+1}, p_{t+1})]$$

s.t

$$c_t = (1 - a_t)m_t$$

$$m_{t+1} = (1 + r)(m_t - c_t) + \text{income}_{t+1}$$

$$\text{income}_{t+1} = \kappa_{t+1}\psi_{t+1}p_{t+1}$$

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

$$a_t \in [0; 1]$$

- ψ_{t+1}, ξ_{t+1} are log-normal shocks.
- Agents retire at age $T^{retired}$ after which they receive low deterministic income.
- **state-vector**: $s_t = (m_t, p_t)$
- **Policy function**: $\pi_t(s_t)$

Applying deep learning

Goal: Solve buffer-stock model while applying deep learning.

- We approximate policy function with DNN: $\pi_t(s_t) \approx \pi^{NN}(t, s_t; \theta_\pi)$
  - θ_π: DNN parameters. Called w_{ij} earlier.
  - Input variables: Time t and state-vector s_t

Applying deep learning

Goal: Solve buffer-stock model while applying deep learning.

- We approximate policy function with DNN: $\pi_t(s_t) \approx \pi^{NN}(t, s_t; \theta_\pi)$
  - θ_π: DNN parameters. Called w_{ij} earlier.
  - Input variables: Time t and state-vector s_t
- Solving the model means finding optimal policy $\pi^{NN}(t, s_t; \theta_\pi) \implies$ Optimal set of parameters θ_π.

Applying deep learning

Goal: Solve buffer-stock model while applying deep learning.

- We approximate policy function with DNN: $\pi_t(s_t) \approx \pi^{NN}(t, s_t; \theta_\pi)$
  - θ_π: DNN parameters. Called w_{ij} earlier.
  - Input variables: Time t and state-vector s_t
- Solving the model means finding optimal policy $\pi^{NN}(t, s_t; \theta_\pi) \Longrightarrow$ Optimal set of parameters θ_π.
- To find optimal θ_π: Maximize some objective function $L(\theta_\pi; \text{input data})$.

How to maximize L

- Deep learning typically uses gradient ascent (or descent):
  $\theta_\pi^{new} = \theta_\pi^{old} + \alpha \nabla_{\theta_\pi} L(\theta_\pi^{old})$

How to maximize L

- Deep learning typically uses gradient ascent (or descent): $\theta_\pi^{new} = \theta_\pi^{old} + \alpha \nabla_{\theta_\pi} L(\theta_\pi^{old})$
- We need the gradient $\nabla_{\theta_\pi} L(\theta_\pi^{old})$.

How to maximize L

- Deep learning typically uses gradient ascent (or descent): $\theta_\pi^{new} = \theta_\pi^{old} + \alpha \nabla_{\theta_\pi} L(\theta_\pi^{old})$
- We need the gradient $\nabla_{\theta_\pi} L(\theta_\pi^{old})$.
- Useful technology: **Automatic differentiation (Autodiff)**
  - If we can compute $L(\theta_\pi^{old}) \implies$ autodiff computes gradient $\nabla_{\theta_\pi} L(\theta_\pi^{old})$.

How to maximize L

- Deep learning typically uses gradient ascent (or descent): $\theta_\pi^{new} = \theta_\pi^{old} + \alpha \nabla_{\theta_\pi} L(\theta_\pi^{old})$
- We need the gradient $\nabla_{\theta_\pi} L(\theta_\pi^{old})$.
- Useful technology: **Automatic differentiation (Autodiff)**
  - If we can compute $L(\theta_\pi^{old}) \implies$ autodiff computes gradient $\nabla_{\theta_\pi} L(\theta_\pi^{old})$.
- Maximization becomes simple: Iteratively compute L while updating θ_π each time using autodiff.

How to maximize L

- Deep learning typically uses gradient ascent (or descent): $\theta_\pi^{new} = \theta_\pi^{old} + \alpha \nabla_{\theta_\pi} L(\theta_\pi^{old})$
- We need the gradient $\nabla_{\theta_\pi} L(\theta_\pi^{old})$.
- Useful technology: **Automatic differentiation (Autodiff)**
  - If we can compute $L(\theta_\pi^{old}) \implies$ autodiff computes gradient $\nabla_{\theta_\pi} L(\theta_\pi^{old})$.
- Maximization becomes simple: Iteratively compute L while updating θ_π each time using autodiff.
- In practice: People typically don't use standard gradient ascent/descent.
  - Instead: ADAM-optimizer - slightly more complicated gradient ascent.

Very simple solution approach: Simulating discounted utility

Instead of using bellman equation:

$$\max_{a_0, a_1, \ldots, a_{T-1}} E_0\Big[\sum_{t=0}^{T-1} \beta^t u(c_t)\Big]$$

$$\text{s.t}$$

$$c_t = (1 - a_t) m_t$$

$$m_{t+1} = (1 + r) m_t + \text{income}_{t+1}$$

$$\text{income}_{t+1} = \kappa_{t+1} \psi_{t+1} p_{t+1}$$

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

$$a_t \in [0; 1]$$

- **Idea**: Maximize $E_0\Big[\sum_{t=0}^{T-1} \beta^t u(c_t)\Big]$ directly.

Setting up the $L(\theta_\pi; \textbf{input data})$

- How to compute $E_0\left[\sum_{t=0}^{T-1} \beta^t u(c_t)\right]$?

Setting up the $L(\theta_\pi; \text{input data})$

- How to compute $E_0\left[\sum_{t=0}^{T-1} \beta^t u(c_t)\right]$?
- Replace expectation with sample average over N agents:

$$L(\theta_\pi; s_{0,i}, \xi_t, \psi_t) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1} \beta^t u(c_t)$$

$$c_t = (1 - m_t)\pi(t, s_t; \theta_\pi)$$

$$m_{t+1} = (1 + r)(m_t - c_t) + \text{income}_{t+1}$$

$$\text{income}_{t+1} = \kappa_{t+1}\psi_{t+1}p_{t+1}$$

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

$$a_t \in [0; 1]$$

Setting up the $L(\theta_\pi; \textbf{input data})$

- How to compute $E_0\Big[\sum_{t=0}^{T-1} \beta^t u(c_t)\Big]$?
- Replace expectation with sample average over N agents:

$$L(\theta_\pi; s_{0,i}, \xi_t, \psi_t) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1} \beta^t u(c_t)$$

$$c_t = (1 - m_t)\pi(t, s_t; \theta_\pi)$$

$$m_{t+1} = (1 + r)(m_t - c_t) + \text{income}_{t+1}$$

$$\text{income}_{t+1} = \kappa_{t+1}\psi_{t+1}p_{t+1}$$

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

$$a_t \in [0; 1]$$

- Input data are initial states $s_{0,i}$ and shocks during lifetime ξ_t and ψ_t.
- Initial states and shocks are exogenous $\implies$ Draw them from distribution we choose.

What is autodiff doing?

In the case where $N = 1$ we can actually write the gradient of the loss-function in an intuitive way:

$$\frac{\partial L}{\partial \theta_\pi} = \frac{\partial a_0}{\partial \theta_\pi} \left(\frac{\partial u_0}{\partial a_0} + \beta \frac{\partial s_1}{\partial a_0} \left(\frac{\partial u_1}{\partial s_1} + \frac{\partial a_1}{\partial s_1} \frac{\partial u_1}{\partial a_1} \right) + \beta^2 \dots \right) - \frac{\partial a_1}{\partial \theta_\pi} \left(\beta \frac{\partial u_1}{\partial a_1} \dots \right) \dots$$

- Very complex gradient.
- Automatic differentation computes this by using chain-rule logic.

Final algorithm

Algorithm:

- For K iterations:
    1. Draw $s_{0,i}$ and $\xi_t, \psi_t \forall t$
    2. Compute $L(\theta_\pi; s_{0,i}, \xi_t, \psi_t)$ by simulation
    3. Update θ_π with gradient ascent and autodiff.

Final algorithm

Algorithm:

- For K iterations:
    1. Draw $s_{0,i}$ and $\xi_t, \psi_t \forall t$
    2. Compute $L(\theta_\pi; s_{0,i}, \xi_t, \psi_t)$ by simulation
    3. Update θ_π with gradient ascent and autodiff.

Is this hard to code?

Final algorithm

Algorithm:

- For K iterations:
    1. Draw $s_{0,i}$ and $\xi_t, \psi_t \forall t$
    2. Compute $L(\theta_\pi; s_{0,i}, \xi_t, \psi_t)$ by simulation
    3. Update θ_π with gradient ascent and autodiff.

Is this hard to code?

- Step 1 is easy.
- Step 2 is the hardest but still just coding a simulation. Typically easy.
- Step 3: standard deep learning code can be used for this.

Why do this instead of dynamic programming?

Some advantages:

- Neural networks are very efficient and flexible interpolators.
- solve the model on simulated grid-points instead of tensor-product grids (see next slides)
- With this specific algorithm: Easy to code.

Why do this instead of dynamic programming?

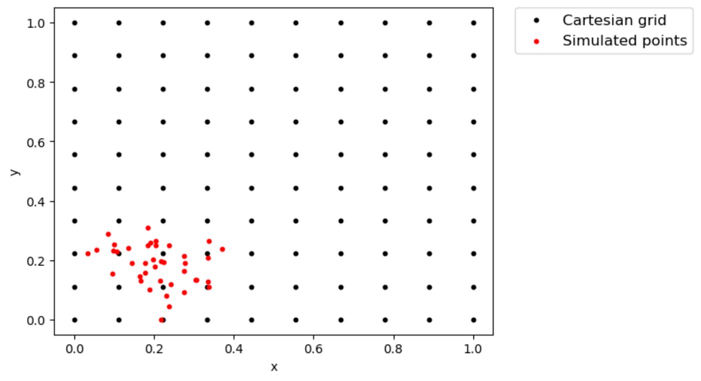
Some advantages:

- Neural networks are very efficient and flexible interpolators.
- solve the model on simulated grid-points instead of tensor-product grids (see next slides)
- With this specific algorithm: Easy to code.

Some disadvantages:

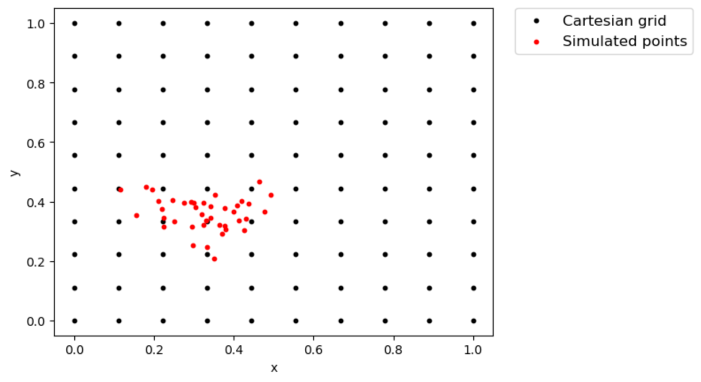
- Neural network slow to train $\implies$ Deep learning inefficient for small models (like the buffer-stock model).
- Hyper-parameters
- No guarantees of convergence

Simulation vs tensor-product grids: iteration 1



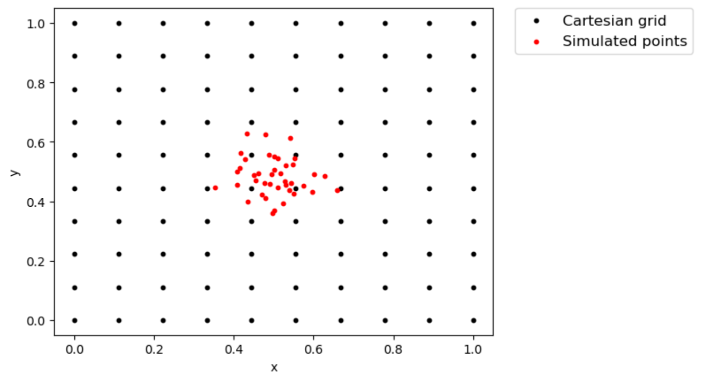
- Initially policy is poor $\Longrightarrow$ and resulting simulated cloud of points is in the wrong area

Simulation vs tensor-product grids: iteration 2



- As policy trains the simulated cloud moves.

Simulation vs tensor-product grids: iteration 3



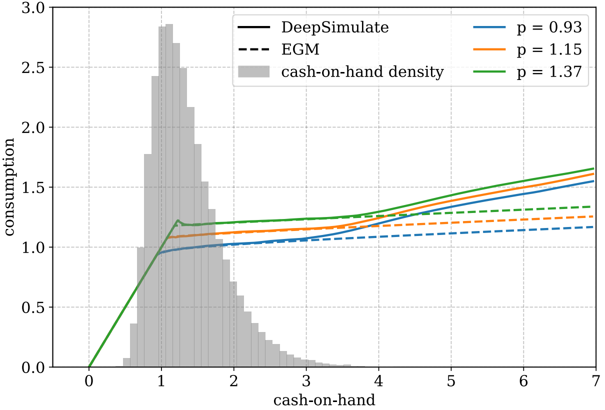
- Eventually it should stabilize.
- **Important point:** Simulated cloud can covers the important part of the state-space with points resulting in us needing less grid-points.

Bufferstock results

Policy functions in $t = 5$

Compare an EGM (DP) consumption policy with the deep learning approach (DeepSimulate in figure).

- Solutions match where we have density!



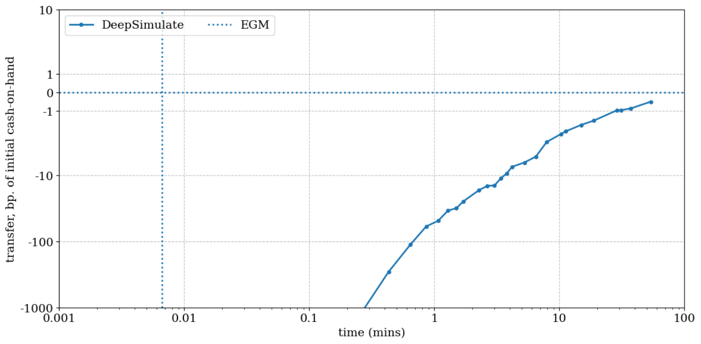
Speed and performance

- Evaluate methods: How do they compare to best DP-methods
- How to compare accuracy?

Speed and performance

- Evaluate methods: How do they compare to best DP-methods
- How to compare accuracy?
- One approach:
    1. Compute avg. discounted utility for each method
    2. convert utility differnences to some monetary measure (similar to consumption equivalents)
- Hardware: DP is on CPU. DL is mainly on GPU.

Simulation approach vs EGM



- **Key Point**: Simulation approach (DeepSimulate) is both slower and less accurate than EGM

Deep Learning in small models

- Deep Learning is inefficient in small models as training neural networks is too costly in terms of time.
- DL only efficient in larger models where curse-of-dimensionality is a huge problem.

Solving a model buffer-stock with too many durables

Buffer-stock model with durables

Goal: Agents choose non-durable consumption c_t and D different durables $d_{j,t}$.

- **States**: Cash-on-hand, persistent income and one for each durable stock.
- **Shocks**: Same as before
- **Inequalities:** Borrowing constraint as before. Also you cannot sell durables.
- **Transaction costs**: Agents face a convex adjustment cost when investing in durables.
- **Number of states**: $2 + D$
- **Number of choices**: $1 + D$
- **Number of inequality constraints**: $1 + D$

Buffer-stock model with durables

$$v_t(m_t, p_t, n_{1,t}, \dots, n_{D,t}) =$$

$$\max_{c_t, d_{1,t}, \dots, d_{D,t}} u(c_t, d_{1,t}, \dots, d_{D,t}) + \beta E_t[v_{t+1}(m_{t+1}, p_{t+1}, n_{1,t+1}, \dots, n_{D,t+1})]$$

s.t

$$\Delta_{t,j} = d_{t,j} - n_{t,j} \geq 0 \quad \forall j \in 1, \dots, D$$

$$m_{t+1} = (1+r)(m_t - c_t - \sum_{j=1}^{D}(\Delta_{j,t} + \Lambda(\Delta_t, j))) + \text{income}_{t+1}$$

$$n_{t+1,j} = (1-\delta_j) n_{t,j} \forall j \in 1, \dots, D$$

$$\text{income}_{t+1} = \kappa_{t+1} \psi_{t+1} p_{t+1}$$

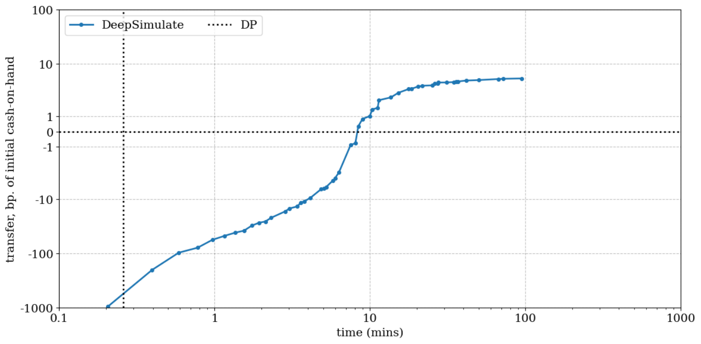
$$p_{t+1} = \xi_{t+1} p_t^{\rho}$$

$$m_t - c_t - \sum_{j=1}^{D}(\Delta_{j,t} + \Lambda(\Delta_t, j)) \geq 0$$

Comparing speed and performance

- We now want to compare speed and performance in this model for a different number of durables D.
- We follow the same accuracy structure as before

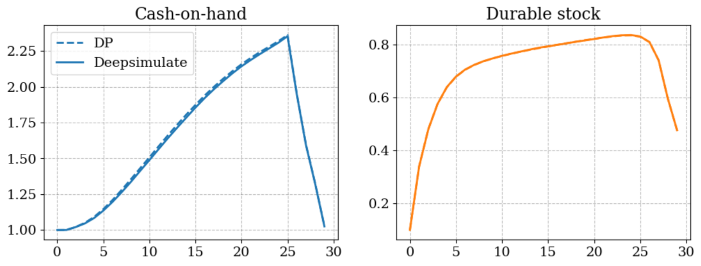
Speed and performance: $D = 1$



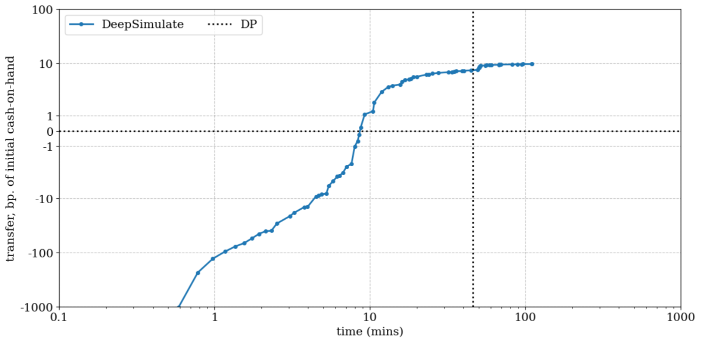
- DL is slower but slightly more accurate than DP
  - Accuracy and speed of DP will depend on number of gridpoints.

Simulated life-cycle profiles: $D = 1$

- Plot features averages
- Both solutions give very similar life-cycle profiles.



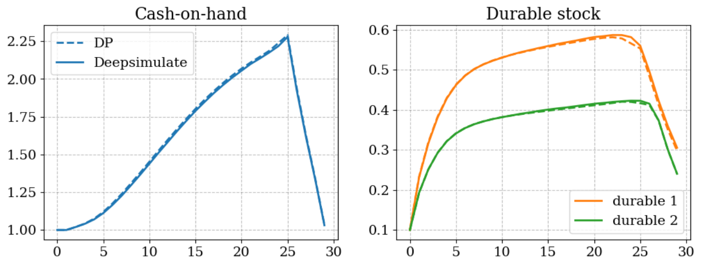
Speed and performance: $D = 2$



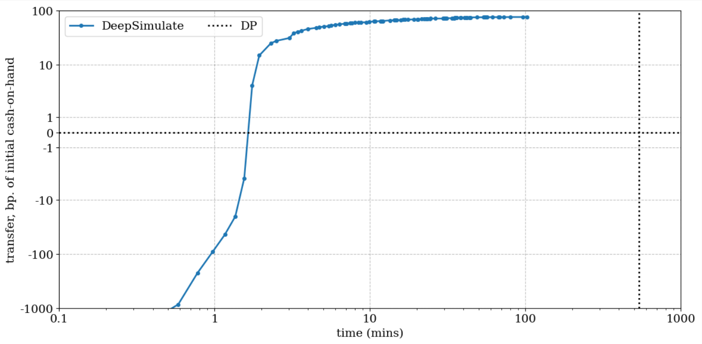
- DL is faster and more accurate than DP
  - Accuracy and speed of DP will depend on number of gridpoints.
  - Possible to specify grid-points such that DP is slightly better but speed will suffer.

Simulated life-cycle profiles: $D = 2$

- Both solutions give very similar life-cycle profiles.



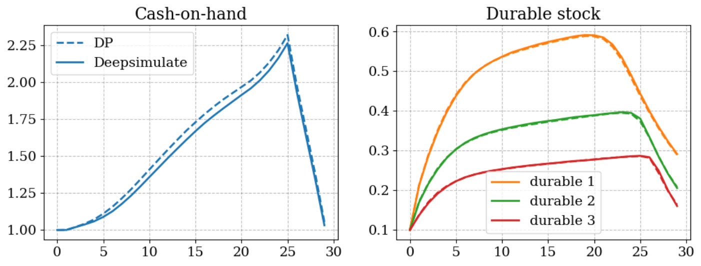
Speed and performance: $D = 3$



- We had to use much fewer gridpoints in DP to solve this within a reasonable amount of time
- DL is much more precise and much faster
- We match the precision of DP after a few minutes.

Simulated life-cycle profiles: $D = 3$

- DP solution seems to get cash-on-hand profile wrong.

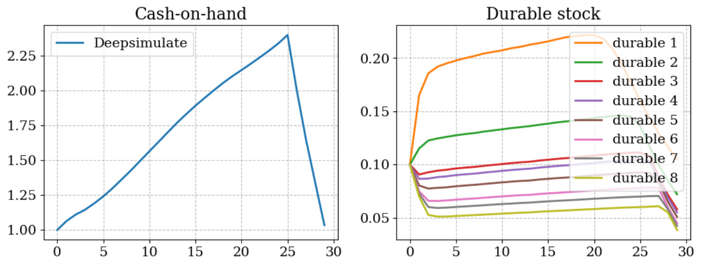


Going to 8 durables

- 8 durables is not doable with dynamic programming
- In the paper we compare several different methods. We check that all methods find the same solution as a sanity check.
- We run the algorithm for four hours.

Simulated life-cycle profiles: $D = 8$

- Life-cycles profiles do not look crazy



Conclusion

- Deep learning and simulation methods can help alleviate the curse of dimensionality.
- There are some problems when using deep learning:
  - Selecting hyper-parameters
  - Stability
  - We might need different estimation approaches.
- Vibrant machine learning community that develops tools that we can use.

Code

Going to the code

- We will use a machine-learning package in python called pytorch.
- I will show you the basic algorithm as described here. Can be improved in a few ways.