# Solving High-Dimensional Dynamic Programming Problems using Deep Learning

Jesús Fernández-Villaverde

University of Pennsylvania, NBER and CEPR

Galo Nuño — Banco de España

George Sorg-Langhans — Princeton University

Maximilian Vogler

Princeton University*

September 22, 2020

**Abstract**

To answer a wide range of important economic questions, researchers must solve high-dimensional dynamic programming problems. This is particularly true in models designed to account for granular data. To break the "curse of dimensionality" associated with these high-dimensional dynamic programming problems, we propose a deep-learning algorithm that efficiently computes a global solution to this class of problems. Importantly, our method does not rely on integral approximation, instead efficiently calculating exact derivatives. We evaluate our methodology in a standard neoclassical growth model and then demonstrate its power in two applications: a multi-location model featuring 50 continuous state variables and a highly nonlinear migration model with 75 continuous state variables.

*Keywords*: Dynamic programming; continuous-time; machine learning; neural networks; granular models.

*JEL codes*: C45, C60, C63.

*The views expressed in this manuscript are those of the authors and do not necessarily represent the views of Banco de España or the Eurosystem. The authors are very grateful to Mark Aguiar, Markus Brunnermeier, Oleg Itskhoki, Nobuhiro Kiyotaki, Ben Moll, Steven Redding, Richard Rogerson, Esteban Rossi-Hansberg, Chris Sims, Gianluca Violante and participants at several seminars for comments. All remaining errors are ours.

# 1 Introduction

To answer a wide range of important economic questions, researchers must solve high-dimensional dynamic programming problems. Examples include consumption-saving problems with many assets, business cycle models with numerous sectors or countries, multiproduct menu-cost models, corporate finance models with various types of capital goods and bonds of different maturities, models of household formation and evolution, models of industry dynamics and production networks, learning models, dynamic games with multiple players, and many others.

In particular, during the last few years, the flourishing of survey evidence and detailed micro data has generated a growing interest in models with granular agents (e.g., models with multiple locations, sectors, goods, etc.) that can incorporate and match this rich observational data. These models with granular agents stand between models with a representative agent and models with a continuum of agents. Up to this point, a fundamental impediment to the study of such granular models has been that the number of states usually grows linearly with the number of agents, as we need to keep track of the individual states of each agent in addition to aggregate states. Thus, even when dealing with just 19 locations (e.g., the 19 member states of the eurozone) or 24 industries (the 24 industry groups in the Global Industry Classification Standard, GICS), we can have dynamic programming problems with dozens, if not hundreds, of state variables.

Solving these high-dimensional dynamic programming problems is exceedingly difficult due to the well-known "curse of dimensionality" (Bellman, 1958, p. ix). As the number of states in the dynamic programming problem grows linearly, the computational burden grows exponentially, both in terms of the number of floating-point operations to be undertaken and regarding memory requirements. While much progress has been made with novel approaches such as sparse grids during the last decade (Brumm and Scheidegger, 2017), solving dynamic programming problems whith over 30 state variables and inherent non-differentiabilities remains an open challenge.

This paper presents a deep-learning algorithm that tackles the "curse of dimensionality" and efficiently provides a global solution to high-dimensional dynamic programming problems. Our approach builds four deep neural networks to approximate i) the value function of the problem, ii) the policy function, and iii) the associated Karush-Kuhn-Tucker multipliers for constraints that hold with equality and inequality. Equivalently, one can think about these four deep neural networks as just one large deep neural network with multiple outputs. However, presenting the algorithm in terms of four separate networks simplifies the exposition.

We plug the four deep neural networks into the Hamilton-Jacobi-Bellman (HJB) equation that formally defines our dynamic programming problem. Then, we build an error criterion by

adding the HJB error, the policy function error, and the constraint error. We train our neural networks by minimizing the error criterion through mini-batch gradient descent over points drawn from the ergodic distribution of the state vector. The whole process is highly efficient and perfectly suitable for parallelization.

Three aspects of our formulation deserve further explanation. First, we use deep neural networks as our approximators. Neural networks are an attractive approach because they are universal nonlinear approximators (Hornik et al., 1989; Cybenko, 1989; Bach, 2017) and break the "curse of dimensionality" (Barron, 1993; Bach, 2017). In comparison, the traditional functional approximation approaches, including polynomials, piecewise polynomials, wavelets, splines, and radial functions, cannot break the "curse of dimensionality."

The intuition is simple: neural networks are compositional (they compose simple activation functions to approximate complex nonlinear functions) while traditional functional approximation methods are additive (they build linear combinations of basis functions). The former approach handles high dimensions much better than the latter because each additional basis function induces changes along all the frequencies of the spectral density of the function to be approximated. In comparison, additional activation functions only move a small range of frequencies, allowing for a fine-tuning of the approximation even in high dimensions.

Among all the different neural network architectures, we use deep neural networks because they have been shown to work surprisingly well in many contexts. Most relevant for us, deep neural networks work very well when dealing with high dimensional PDEs, a class of mathematical problems that includes dynamic programming (Han et al., 2018; Sirignano and Spiliopoulos, 2018). However, our approach could easily accommodate alternative architectures.

Second, we can simulate training data directly from the underlying model given the current state of the neural networks. This obviates the need for a validation and training set as well as regularization techniques designed to reduce overfitting. Whilst in practice we train our model on more than one billion data points to achieve convergence, in principle we could generate arbitrary amounts of data.

Third, we cast our problem in continuous time. Although our algorithm applies to discrete time with minor changes of notation, our formulation in continuous time offers one key advantage: we get rid of integrals. An often overlooked aspect of the "curse of dimensionality" is the need to evaluate high-dimensional integrals in the continuation value function.

The relevant set to evaluate an integral is the so-called typical set of the distribution of interest (Cover and Thomas, 2012). The typical set of high-dimensional distributions feature two crucial but surprising properties. First, in general, the typical set is not the region where the associated density is the highest. Thus, naively searching for quadrature points or simulating around that high-density region will result in low accuracy in the approximation of the integrals. Second, by concentration of measure, the typical set will be a narrower and narrower band as

the number of states grows. Thus, in most situations with high dimensions, we will waste nearly all computations while approximating integrals. Unless we have information about where this typical set lays, we will be dealing with regions of the distribution that are (nearly) irrelevant for the computation of the integral.

Unfortunately, characterizing the typical set is hard. Algorithms that do so, such as the Hamiltonian Monte Carlo are costly to implement (see, for details, Fernández-Villaverde and Guerrón-Quintana, 2020). A much more fruitful approach is to cast the dynamic programming problem in continuous time, taking advantage of Itô's Lemma, and therefore dropping the integrals. Computing the derivatives implied by Itô's Lemma is considerably easier than computing any integral, in particular given our neural network architecture and the properties of the backpropagation algorithm used for gradient computation.

Having said that, we reiterate that our algorithm can be applied to discrete time problems and keep all of its advantages with respect to traditional functional approximation procedures (all of which suffer from the exact same need of evaluating high-dimensional integrals).

We illustrate our algorithm with three applications. First, we solve a standard neoclassical growth model with a conventional calibration. Computationally, this is a straightforward problem but it provides us with a sharp testbed to show how our method works and gauge its accuracy.

Next, we show how to solve a dynamic capital allocation model with our algorithm. The model investigates optimal consumption, investment, and cross-location capital flows in a model of aggregate productivity shocks and capital movement frictions. We calibrate the model to 25 locations. There are two continuous state variables in each location, which generate a total of 50 continuous aggregate state variables.

Finally, we turn to a setting of inherent non-linearity in order to illustrate the global solution properties of our method. Specifically, we study the propagation of economic shocks in a multi-country model featuring fully rational expectations, migration linkages, and 75 continuous aggregate state variables.

Our paper is related to the contributions of Maliar et al. (2019) and Azinović et al. (2019), who propose discrete-time algorithms. While similar in spirit in their use of neural networks as global approximators, these papers face the hurdle of approximating the expectations integral as explained above. Our approach elegantly circumvents this issue by using the analytic solutions provided by the continuous-time limit. Another related approach to ours is based on the reinforcement learning literature in the spirit of Sutton and Barto (2018) as applied by Duarte (2018) in a continuouts-time context. In contrast to this approach, we utilize the fact that in economic problems we have additional knowledge of the underlying problem structure. Specifically, we can draw on the first-order conditions associated with the problem to train our neural networks. This additional information helps us greatly in pinning down the policy functions

and extending the dimensionality of the problems we can solve.

The rest of the paper is organized as follows. Section 2 introduces our deep learning algorithm, discusses some of its most salient aspects, and compares it with other related methods. Section 3 presents our first application: the standard neoclassical growth model. Section 4 describes our second application: a dynamic capital allocation model. Section 5 introduces our third application: a migration model. Section 6 concludes.

# 2 Solution Method

We now present the details of our solution algorithm. We start with an outline of the generic dynamic programming problem we want to handle and discuss the challenges associated with its solution. Next, we introduce our deep-learning algorithm, followed by an in-depth discussion of its elements. We close this section by discussing the differences of out approach to alternative algorithms recently proposed in the literature.

## 2.1 Outline of Problem

To start with, we standardize notation for the rest of the paper. We will denote scalar values as lower case letters, vectors as bold lower case letters, and matrices as bold upper case letters. Vectors are column vectors. Thus, transposed vectors indicate row vectors.

Our goal is to solve the recursive continuous-time HJB equation globally:

$$\begin{aligned} \rho V(\mathbf{x}) = \max_{\boldsymbol{\alpha}} r(\mathbf{x}, \boldsymbol{\alpha}) + \nabla_x V(\mathbf{x}) f(\mathbf{x}, \boldsymbol{\alpha}) + \frac{1}{2} tr(\sigma(\mathbf{x}))^T \Delta_x V(\mathbf{x}) \sigma(\mathbf{x})) \\ \text{s.t.} \quad G(\mathbf{x}, \boldsymbol{\alpha}) \leq \mathbf{0} \quad \text{and} \quad H(\mathbf{x}, \boldsymbol{\alpha}) = \mathbf{0}, \end{aligned} \tag{1}$$

where $V : \mathbb{R}^N \to \mathbb{R}$ is a value function depending on a state vector $\boldsymbol{x} \in \mathbb{R}^N$ with $N \in \mathbb{R}$, $\nabla_x V(\mathbf{x}) \in \mathbb{R}^N$ is the gradient of $V$, and $\Delta_x V(\mathbf{x}) \in \mathbb{R}^{N \times N}$ is the Hessian of $V$.

In problem (1), $\rho$ denotes the discount rate, $r : \mathbb{R}^{N \times M} \to \mathbb{R}$ is the instantaneous return function, that depends on the states and the control vector $\boldsymbol{\alpha} \in \mathbb{R}^M$, where $M \in \mathbb{R}$. The state vector follows a diffusion process with drift $f : \mathbb{R}^{N \times M} \to \mathbb{R}^N$ and diffusion $\sigma : \mathbb{R}^N \to \mathbb{R}^N$. The problem is constrained by the constraint functions $G : \mathbb{R}^M \to \mathbb{R}^{L_1}$ and $H : \mathbb{R}^M \to \mathbb{R}^{L_2}$, where $L_1, L_2 \in \mathbb{R}$ are the number of inequality and equality constraints respectively. We denote by $\alpha : \mathbb{R}^N \to \mathbb{R}^M$ the policy function, which is the maximized control vector given some states (notice the absence of bold font to distinguish the policy function from the control vector itself).

We are interested in solving the problem (1) when $N$ is high, that is when we are facing a highly-dimensional problem. It is well known that, in this case, problem (1) suffers from an acute "curse of dimensionality." Grid-based approaches quickly become infeasible since

the number of required grid points grows exponentially with $N$ for a given precision. Local solution methods such as linearization or higher-order polynomial expansions are not suitable for problems that display irregular behavior, such as kinks or other strong nonlinearities.

In contrast, our solution algorithm copes well with both of those challenges. Our approach is capable of i) globally approximating irregularly behaved solutions to ii) highly-dimensional problems.

## 2.2 Solution Algorithm

Our approach to solving problem (1) buils on the ideas of deep learning. We use neural networks as global nonlinear approximators for both value and policy functions. We define an error criterion, which is composed of the HJB error and the first-order conditions (FOC) errors. Then, we train the neural networks (i.e., update the weights in the networks) by picking points in the state space from their ergodic distribution and minimize our error criterion using mini-batch gradient descent until convergence. Let us consider each of these steps in detail.

First, we define four neural networks $\tilde{V}(\mathbf{x};\boldsymbol{\Theta}^V) : \mathbb{R}^N \to \mathbb{R}$, $\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha) : \mathbb{R}^N \to \mathbb{R}^M$, $\tilde{\mu}(\mathbf{x};\boldsymbol{\Theta}^\mu) : \mathbb{R}^N \to \mathbb{R}^{L_1}$, and $\tilde{\lambda}(\mathbf{x};\boldsymbol{\Theta}^\lambda) : \mathbb{R}^N \to \mathbb{R}^{L_2}$. These four neural networks are parameterized by weights $\boldsymbol{\Theta}^V$, $\boldsymbol{\Theta}^\alpha$, $\boldsymbol{\Theta}^\mu$, and $\boldsymbol{\Theta}^\lambda$ to approximate i) the value function $V(\mathbf{x})$, ii) the policy function $\boldsymbol{\alpha}$, and Karush-Kuhn-Tucker (KKT) multipliers iii) $\boldsymbol{\mu}$ and iv) $\boldsymbol{\lambda}$. To simplify notation, we accumulate all neural network weights in the matrix $\boldsymbol{\Theta} = (\boldsymbol{\Theta}^V, \boldsymbol{\Theta}^\alpha, \boldsymbol{\Theta}^\mu, \boldsymbol{\Theta}^\lambda)$.[1] If the reader is unfamiliar with neural networks, it suffices for now to treat a neural network as any parameterized function approximator. We will provide a brief exposition along with a discussion of its advantages for this problem below.

Second, we define our error criterion. It is composed of three elements: the HJB error, the policy function error, and the constraint error. For any point $\mathbf{x} \in \mathbb{R}^N$ in the state space, the HJB error is defined as the difference between the right and the left-hand sides of the HJB when we substitute the exact value and policy functions by their approximations:

$$err_{HJB}(\mathbf{x};\boldsymbol{\Theta}) \equiv r(\mathbf{x}, \tilde{\alpha}(\mathbf{s};\boldsymbol{\Theta}^\alpha)) + \nabla_x \tilde{V}(\mathbf{x};\boldsymbol{\Theta}^V) f(\mathbf{x}, \tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha)) +$$
$$+ \frac{1}{2} tr[\sigma(\mathbf{x})^T \Delta_x \tilde{V}(\mathbf{x};\boldsymbol{\Theta}^V)\sigma(\mathbf{x})] - \rho \tilde{V}(\mathbf{x};\boldsymbol{\Theta}^V)$$

Analogously, the policy function error is defined as the difference of the FOC from zero when

[1] We will reserve the term "parameter" for elements defining the economic problem (e.g., risk aversion, discount rate) and "weights" for the elements that parameterize the neural network but that, in general, do not have any sharp economic interpretation.

we plug in the approximated value and policy functions:

$$err_\alpha(\mathbf{x};\boldsymbol{\Theta}) \equiv \frac{\partial r(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha)}{\partial \boldsymbol{\alpha}} + D_\alpha f(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha))^T \nabla_x \tilde{V}(\mathbf{x};\boldsymbol{\Theta}^V) \\ - D_\alpha G(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha))^T \tilde{\mu}(\mathbf{x};\boldsymbol{\Theta}^\mu) - D_\alpha H(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha))\tilde{\lambda}(\mathbf{x};\boldsymbol{\Theta}^\lambda),$$

where $D_\alpha G \in \mathbb{R}^{L_1 \times M}$, $D_\alpha H \in \mathbb{R}^{L_2 \times M}$, and $D_\alpha f \in \mathbb{R}^{N \times M}$ are the submatrices of the Jacobian matrices of $G$, $H$ and $f$ respectively containing the derivatives with respect to $\alpha$.

Finally, the constraint error is itself composed of four different errors: the primal feasibility errors:

$$err_{PF_1}(\mathbf{x};\boldsymbol{\Theta}) \equiv \max\{0, G(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha))\} \qquad err_{PF_2}(\mathbf{x};\boldsymbol{\Theta}) \equiv H(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha)),$$

the dual feasibility error:

$$err_{DF}(\mathbf{x};\boldsymbol{\Theta}) = \max\{0, -\tilde{\mu}(\mathbf{x};\boldsymbol{\Theta}^\mu\},$$

and the complementary slackness error:

$$err_{CS}(\mathbf{x};\boldsymbol{\Theta}) = \tilde{\mu}(\mathbf{x};\boldsymbol{\Theta})^T G(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^\alpha)).$$

We combine these four errors by using the squared error as our loss criterion:

$$\mathcal{L}(\mathbf{x};\boldsymbol{\Theta}) \equiv \left|\left|err_{HJB}(\mathbf{x};\boldsymbol{\Theta})\right|\right|_2^2 + \left|\left|err_\alpha(\mathbf{x};\boldsymbol{\Theta})\right|\right|_2^2 + \left|\left|err_{PF_1}(\mathbf{x};\boldsymbol{\Theta})\right|\right|_2^2 + \\ + \left|\left|err_{PF_2}(\mathbf{x};\boldsymbol{\Theta})\right|\right|_2^2 + \left|\left|err_{DF}(\mathbf{x};\boldsymbol{\Theta})\right|\right|_2^2 + \left|\left|err_{CS}(\mathbf{x};\boldsymbol{\Theta})\right|\right|_2^2 \tag{2}$$

We train our neural networks by minimizing the above error criterion through mini-batch gradient descent over points drawn from the ergodic distribution of the state vector.

The efficient implementation of this last step is the key to the success of our algorithm. We start by initializing our network weights (a detailed discussion of how to pick good initial weights is provided in the appendix). Then, we perform $K$ learning steps called epochs, where $K$ can be chosen in a variety of ways discussed in the appendix. For each epoch, we draw $I$ points from the state space according to its ergodic distribution. This is computationally very inexpensive when a closed-form expression for the ergodic distribution is available and still relatively inexpensive if the ergodic distribution has to be simulated. Then, we randomly split this sample into $B$ mini-batches of size $S$. For each mini-batch, we define the mini-batch error, by averaging the loss function over the batch. Finally, we perform mini-batch gradient descent for all network weights, with $\eta_k$ being the learning rate in the $k$-th epoch, the choice of which is again discussed in the appendix.

Algorithm 1 provides an overview over our entire algorithm.

**Algorithm 1:** Deep-Learning Solution Algorithm for HJB

Parameterize value and policy functions through neural networks $\tilde{V}$, $\tilde{\alpha}$, $\tilde{\mu}$, and $\tilde{\lambda}$;

Define the loss function $\mathcal{L}$ as in (2);

Initialize neural network weights $\boldsymbol{\Theta}_1$;

**for** $k = 1, ..., K$ **do**

  Draw $I$ sample points $\{\mathbf{x}_i\}_{i=1}^I$ from their ergodic distribution $\nu(\mathbf{x})$;

  Randomly split these points into $B$ mini-batches of size $S$: $\{\{\mathbf{x}_{s,b}\}_{s=1}^S\}_{b=1}^B$;

  Use previous weights as starting weights for new step: $\boldsymbol{\Theta}_{k,1} = \boldsymbol{\Theta}_k$;

  **for** $b = 1, ..., B$ **do**

    Define batch error: $\bar{\mathcal{L}}(\{\mathbf{x}_{s,b}\}_{s=1}^S; \boldsymbol{\Theta}_{k,b}) \equiv \frac{1}{S}\sum_{s=1}^S \mathcal{L}(\mathbf{x}_{s,b}; \boldsymbol{\Theta}_{k,b})$;

    Train $\tilde{V}$: $\boldsymbol{\Theta}^V_{k,b+1} = \boldsymbol{\Theta}^V_{k,b} - \eta_k \nabla_{\boldsymbol{\Theta}^V_{k,b}} \bar{\mathcal{L}}(\{\mathbf{x}_{s,b}\}_{s=1}^S; \boldsymbol{\Theta}_{k,b})$;

    Train $\tilde{\alpha}$: $\boldsymbol{\Theta}^\alpha_{k,b+1} = \boldsymbol{\Theta}^\alpha_{k,b} - \eta_k \nabla_{\boldsymbol{\Theta}^\alpha_{k,b}} \bar{\mathcal{L}}(\{\mathbf{x}_{s,b}\}_{s=1}^S; \boldsymbol{\Theta}_{k,b})$;

    Train $\tilde{\mu}$: $\boldsymbol{\Theta}^\mu_{k,b+1} = \boldsymbol{\Theta}^\mu_{k,b} - \eta_k \nabla_{\boldsymbol{\Theta}^\mu_{k,b}} \bar{\mathcal{L}}(\{\mathbf{x}_{s,b}\}_{s=1}^S; \boldsymbol{\Theta}_{k,b})$;

    Train $\tilde{\lambda}$: $\boldsymbol{\Theta}^\lambda_{k,b+1} = \boldsymbol{\Theta}^\lambda_{k,b} - \eta_k \nabla_{\boldsymbol{\Theta}^\lambda_{k,b}} \bar{\mathcal{L}}(\{\mathbf{x}_{s,b}\}_{s=1}^S; \boldsymbol{\Theta}_{k,b})$;

  **end**

  Update step weights: $\boldsymbol{\Theta}_{k+1} = \boldsymbol{\Theta}_{k,B+1}$;

**end**

## 2.3 Discussion

It is worthwhile to discuss the individual elements of the algorithm. In particular, we need to explain the advantages of formulating our problem in continuous-time, the choice of neural networks as our global nonlinear approximator, the updating algorithm, and the generation of training data. We conclude this section by highlighting the main differences in training neural networks on model-generated data compared to their traditional application to real-world generated data.

### 2.3.1 Advantages of Continuous Time

One crucial element for our algorithm is that we formulate problem (1) in continuous time. Technically, this means that we have to handle partial differential equations instead of partial difference equations. In comparison with the existing neural network algorithms existing

in the literature, framed in discrete time, the continuous-time formulation enjoys one crucial advantage.

To understand why this is the case, consider the discrete-time value function analogous to the continuous-time HJB, which has the following form:

$$V(\mathbf{x}) = \max_{\boldsymbol{\alpha}} r(\mathbf{x}, \boldsymbol{\alpha}) + \beta \mathbb{E}[V(\mathbf{x}')] \tag{3}$$

Notice that now the continuation value includes an expectation term, i.e., an integral over all possible realizations of $\mathbf{x}'$. This is a fundamental obstacle for solution algorithms, since for each point of the state space, one needs to approximate this integral –a potentially time-expensive and inaccurate step. In particular, the solution algorithm needs to know (or cover) the typical set, the relevant region of the state variables for the computation of the expectation of $\mathbf{x}$.

For $\epsilon > 0$ and any $I$, we define the typical set $A_\epsilon^I$ with respect to a density $p(\mathbf{x})$ as:

$$\left\{ (\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_I) : \left| -\frac{1}{I+1} \sum_{i=0}^{I} \log p(\mathbf{x}_i) - h(\mathbf{x}) \right| \leq \epsilon \right\},$$

where $h(\mathbf{x}) = -\int p(\mathbf{x}_i) \log p(\mathbf{x}_i) d\mathbf{x}$ is the differential entropy of the vector $\mathbf{x}$ with respect to its density (a measure of the concentration of a density; Cover and Thomas, 2012, ch. 8).

By a weak law of large numbers, $Pr(A_\epsilon^I) > 1 - \epsilon$ for $I$ sufficiently large. In other words, we call $A_\epsilon^I$ "typical" because it includes "most" sequences of $\mathbf{x}_i$'s that are distributed according to $p(\mathbf{x})$. This result shows why $A_\epsilon^I$ is, indeed, the relevant region to compute expectations of $\mathbf{x}$ or of functions of $\mathbf{x}$.

Two properties of the typical set are counterintuitive, but crucial. First, in general, the typical set is not the region where the density $p(\mathbf{x})$ is the highest. Second, by concentration of measure, the typical set will be a narrower and narrower band as the dimensionality of $\mathbf{x}$ grows.

Since, in general, we will not know where the typical set of the density of the state $\mathbf{x}$ is, we will either spend too many resources computing the integral in the Bellman operator or get an inaccurate evaluation of the expectation. Additionally, this challenge becomes exponentially worse as the number of state variables with respect to which we need to take the integral grows.

As shown in the appendix, the discrete-time problem (3) converges towards the HJB problem (1) as the time step $\Delta t \to 0$. But, importantly, once we reach continuous time, we do not face the integral-approximation issue, since due to the application of Ito's Lemma, we get an analytical expression for this expectation, which depends only on the gradient and Hessian of $V$. Furthermore, the efficient computation of gradients and Hessians is one of the key advantages of neural networks.

In summary, jumping to continuous time allows us to substitute a challenging problem, the evaluation of integrals, for a much simpler one, the computation of gradients and Hessians with

neural networks training algorithms.

### 2.3.2 Neural Networks

**Definition** (Artificial) neural networks are a class of nonlinear functions, mapping $\mathbb{R}^N \to \mathbb{R}^J$, that have gained much popularity over the past two decades thanks to breakthroughs in training algorithms (Rumelhart et al., 1986; Glorot and Bengio, 2010) as well as the vast increase in the availability of both data and computing power.

We will work with a simple version of a neural network, namely a multi-layer feedforward network. In the appendix, we discuss more complex neural network structures such as convolutional networks and recursive neural networks such as Long Short-Term Memory (LSTM) networks. Also, for simplicity, we will present the case where the final outcome of the neural network is a single output ($J = 1$), but the approach is easy to generalize to multiple outputs ($J > 1$). In fact, we use that idea below where we create layers.

At the lowest level a neural network is composed of a"neuron," which is a function $\mathbb{R}^N \to \mathbb{R}$ of the form:

$$m(\mathbf{x}; \boldsymbol{\theta}) \equiv \phi\left(\theta_0 + \sum_{i=1}^{N} \theta_i x_i\right)$$

The function takes an input $\mathbf{x} \in \mathbb{R}^N$ and is parameterized by the weight vector $\boldsymbol{\theta} \in \mathbb{R}^{N+1}$, which contains both the signal weights $\{\theta_i\}_{i=1}^N$ and the bias $\theta_0$. The activation function $\phi(\cdot)$ is an appropriate nonlinear function, for example the hyperbolic tangent $\phi(x) = \tanh(x)$ or the rectifier $\phi(x) = \max\{0, x\}$. Intuitively, a neuron is just a nonlinear transformation of a linear combination of its inputs.

We can create a "layer" by stacking $N_1$ neurons on top of each other. That is, a layer is a function $\mathbb{R}^N \to \mathbb{R}^{N_1}$:

$$M(\mathbf{x}; \tilde{\boldsymbol{\Theta}}) \equiv (m(\mathbf{x}; \boldsymbol{\theta}_1), ..., m(\mathbf{x}; \boldsymbol{\theta}_{N_1}))^T,$$

where $\tilde{\boldsymbol{\Theta}}$ is the matrix of all parameters of the layers, given by $\tilde{\boldsymbol{\Theta}} = (\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_{N_1})$. All neurons in a layer share the same input $\mathbf{x}$. However, since each neuron $i$ has a different parameterization $\boldsymbol{\theta}_i$, this yields $N_1$ different outputs of the layer.

A neural network now combines multiple such layers, by feeding the output of the previous layer as the input into the next layer.[2] Finally, the output of the last layer is fed into an output layer. This is a single neuron with an identity activation function, i.e. it is a simple linear

[2]To improve performance, the inputs into the first layer are usually normalized, a detailed discussion of which can be found in the appendix.

combination of its inputs.[3] A neural network is therefore the composition of different layers. For example a neural network with two layers is a function $\mathbb{R}^N \to \mathbb{R}$:

$$NN(\mathbf{x}; \mathbf{\Theta}) \equiv \theta_0^{Out} + \boldsymbol{\theta}_{1:N_2}^{Out} M(M(\mathbf{x}; \tilde{\mathbf{\Theta}}_1); \tilde{\mathbf{\Theta}}_2)),$$

where $N_2$ is the number of output neurons of the second layer and $\mathbf{\Theta}$ is the matrix of all parameters of the neural network, comprised of the parameters of all hidden layers as well as of the output layer $\mathbf{\Theta} = (\tilde{\mathbf{\Theta}}_1, \tilde{\mathbf{\Theta}}_2, \boldsymbol{\theta}^{Out})$, with $\boldsymbol{\theta}^{Out} = (\theta_0^{Out}; \boldsymbol{\theta}_{1:N_2}^{Out})$. Since the outputs of all layers except for the output layer are generally not of direct interest, those layers are commonly referred to as "hidden layers". A neural network with more than two hidden layers is called a "deep" neural network.

A visualization of such a network with two hidden layers is depicted in figure 1. Here, we feed three inputs which are represented by the three red circles in the input layer into the network. All inputs are then fed into each neuron in the first hidden layer. Each neuron in this layer nonlinearly combines these inputs into one output as described above. All of those outputs are then in turn fed as inputs into the neurons of the second hidden layer, which again transform them. Finally, all the outputs of the second hidden layer are combined into one final output, represented by the green circle in the output layer.

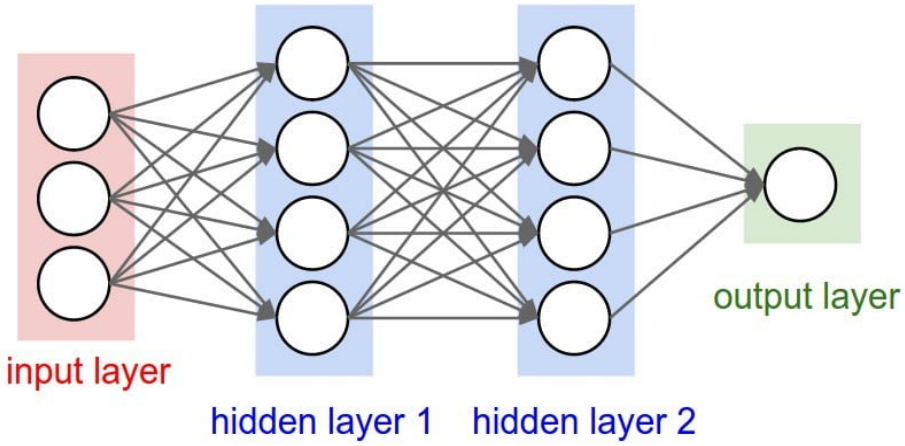



Figure 1: A neural network with two hidden layers.

Note that neural networks do not necessarily have to only have one output, they can also be a more general mapping $\mathbb{R}^N \to \mathbb{R}^M$. In this case the output layer itself has multiple neurons, each producing a different output. We will use such multi-output networks in our applications.

[3] The last layer does not necessarily have to use an identity activation function. Depending on the application, we can use different activation functions for the output layer. For example, logistic or softmax functions are frequently used for classification tasks.

**Neural Networks as a Global Nonlinear Approximator** Besides neural networks, there exists a wide array of alternative classes of approximators. For example, we could have chosen Chebyshev, Gegenbauer, or Legendre polynomials, of which Chebyshev polynomials are most widely used in economics.[4] It is well known that for Chebyshev polynomials the approximation error becomes arbitrarily small as the number of degrees goes to infinity (Erdös and Turan, 1937), and that said error can be approximated for finite degrees (Boyd, 2001). However, an equivalent theorem exists for neural networks. The Universal Approximation Theorem (Hornik et al., 1989; Cybenko, 1989; Bach, 2017) shows that a neural network with at least one layer can approximate any Borel-measurable function of finite dimensionality to any degree of accuracy.

The main theoretical strength of neural networks, however, stems from a theorem by Barron (1993). It states that under some technical conditions a one-layer neural network achieves integrated square errors of order $O(1/n)$, where $n$ is the number of neurons. In comparison, for series approximations, such as polynomials, splines, or trigonometric expansions, the integrated square error is of order $O(1/(n^{2/d}))$, with $d$ denoting the number of dimensions of the function to be approximated.[5] In other words, holding the approximation error fixed, the number of neurons needed to approximate the function grows linearly with the number of dimensions for neural networks, whilst it grows exponentially for series approximations. This theorem is of fundamental importance for our focus on high-dimensional problems, since it essentially states that the "curse of dimensionality" does not apply to the same extent to neural networks. Note that this theorem makes no statement about the intercept, so it is well possible that for low dimensions series approximations require fewer terms. However, for the high dimensions that we are interested in neural networks are the best approximators.

Note that this power does not come for free. Neural networks have very high data requirements compared to other approximators, which has prevented them from being in widespread use until the recent rise of big data. However, for our algorithm we have in principle access to an arbitrarily large amount of data as we will discuss below. Nevertheless, this data requirement stresses that our algorithm is tailored towards high-dimensional problems and does not aim to compete with alternatives on low-dimensional problems.

In addition to these theoretical strengths, neural networks have also achieved great empirical success over a wide range of problems. They have been observed to extrapolate significantly better beyond their training set than alternatives such as Chebyshev polynomials (for an economic illustration see Fernández-Villaverde et al., 2019).

The final strength of neural networks is the relative ease and efficiency with which they can be estimated. As noted above, our algorithm requires an extremely efficient computation of

[4] They appear to be the best suited for the conventional small state space problems found in economics, as discussed by Boyd and Petschek (2014).

[5] These results are extended by Bach (2017) to cover nondecreasing positively homogeneous activation functions, such as the rectifier, and to derive approximation and estimation errors.

neural network gradients with respect to both its parameters $\Theta$ needed for the gradient descent step and the states $\mathbf{x}$ required for the computation of the loss function. The backpropagation algorithm discussed in the appendix provides an extremely efficient way to compute all those gradients for neural networks (Rumelhart et al., 1986). In particular, it can be shown that the computation of all gradients requires the same computation time as one evaluation of the neural network itself (Baydin et al., 2018).

### 2.3.3 Mini-batch Gradient Descent

As alluded to above, one crucial advantage of neural networks is that basic gradient-descent algorithms yield efficient and reliable updates. In our algorithm, we have opted for mini-batch gradient descent as the basic training algorithm. Let us briefly discuss this decision.

Fundamentally, there are two different approaches for gradient descent. First, one could perform one training step per epoch, with the loss function averaged over all $I$ data points generated in one epoch, an approach known as deterministic gradient descent. Second, one could have one training step per data point, which is referred to as stochastic gradient descent. The rationale for deterministic gradient descent is that the large number of points used in the average will lead to a very good approximation of the real gradient, so that the algorithm will always follow the steepest slope down the error surface. Whilst this works well if the error surface is convex, this is problematic if it is less well-behaved as the algorithm might get stuck in local minima. Stochastic gradient descent on the other hand, only poorly approximates the gradient with one observation, which slows down its learning. On the other hand, it is able to cope with local minima, as the poor approximation can catapult the algorithm out of local minima. In line with most of the modern neural network literature, we use mini-batch gradient descent as a compromise between these two extremes, which exhibits both fast convergence and robustness to local minima.

Note that we have written our algorithm in terms of plain mini-batch gradient descent for simplicity of exposition. In our code, we use momentum-based updating algorithms, which are more robust than mini-batch gradient descent. However, since these algorithms are based on mini-batch gradient descent, the underlying logic and challenges are the same, so that we delegate a discussion of the optimizing details to the appendix.

At this point the reader might wonder why we split the simulated data into different mini-batches instead of only simulating the number of points needed in the current batch. The rationale for this is that the simulation step often comes with a fixed cost, like. a burn-in period described below, so that it is computationally advantageous to simulate large amounts of data in one step.

### 2.3.4 Generation of Training Data

One crucial element of our algorithm is to draw our training points from their ergodic distribution over the state space. In particular, the ergodic distribution of the state variables in many economic models has mass over only a small subspace of the entire state-space hypercube. We can thus focus our algorithm on this small subspace, enabling us to shrink the hypercube to a significantly smaller subspace. This therefore simplifies the problem for our neural-network approximators as their training points are concentrated in a smaller space, making it easir to approximate.

Importantly, this reduction in complexity can be achieved at a very low cost. In particular the ergodic distribution can be simulated based on the current approximation of policy functions. This simulation requires only the evaluation of the policy neural networks, which can be done very efficiently, so that this is again an inexpensive process. For every epoch we can thus simulate a large number of points at low cost.[6]

To maximize the probability that our draws are actually made from the ergodic distribution, we use the following simulation procedure. For the first $K_1$ epochs, we draw points uniformly at random from the entire hypercube. This is necessary since in the beginning the neural networks are still ill-converged, so using them for simulation could possibly lead us to pick points only from far-out regions of the state-space. For every subsequent epoch we then draw $I$ random starting points uniformly at random from the state-space hypercube This is important to overcome potential local convergence issues. From each of those points we iterate $T$ steps forward. We drop the first $B_1$ iterations as a burn-in, since the starting points are random and might thus be outside of the ergodic distribution. After $B$ iterations, it is likely that the simulation has converged towards the ergodic distribution. We keep all remaining iterations, shuffle them randomly and use the resulting set as our training point for the kth epoch. Note that it is important to draw only a limited number of points per epoch so that updates of the ergodic distribution based on updates of the current approximation are quickly incorporated. We provide a detailed exposition of this procedure for one of our applications in the appendix.

### 2.3.5 Training on Model-Generated Data

Finally, we will discuss differences between our approach and more traditional neural network applications. In particular, readers familiar with neural networks might have noted the absence of training, validation, and test samples as well as the absence of a discussion of overfitting.[7]

[6] Note that in some problems the ergodic distribution can even be expressed through analytical expressions, in which case points can be drawn from it even more easily.

[7] Overfitting is a common problem when training on a finite data set, where the network might fit too well to its training data. By doing so, it concentrates more on fitting the individual data points than finding the underlying structure, leading to a weak predictive power beyond the training set.

The reason for this is that we train our neural network on model-generated data, which means that in principle we can generate arbitrarily large amounts of data to train our model. As shown in the above algorithm, this enables us to use each data point only once in the training process, so that we do not need to be overly concerned about overfitting. In other words, since the neural network does not repeatedly face the same training data, it is impossible for it to fit every individual data point. The absence of the overfitting problem thus also obviates the need for a validation and test set, which are used to alleviate said problem.

Another consequence of this is that regularization is not as important in our context as in data-science applications. We have experimented with a wide array of regularization techniques, but as expected this did not lead to increase in accuracy but a slight increase in running time.

Beyond this crucial difference, our approach is very similar to data-science applications of neural networks - we train our neural networks using our model-generated data on a carefully-designed loss function.

## 2.4 Comparison to Related Methods

Recently, there has been an increased interest in the use of deep-learning techniques to solve large-state space problems globally. First, recent contributions by Maliar et al. (2019), as well as by Azinović et al. (2019) propose discrete-time algorithms. While similar in spirit in their use of neural networks as global approximators, they face the hurdle of approximating the expectations integral as explained above. Our approach elegantly circumvents this issue by using the analytic solutions provided by the continuous-time limit.

Next, Sirignano and Spiliopoulos (2018) propose a related algorithm in the applied mathematics literature. They are concerned with the general solution to partial differential equations in a wide range of applications. Compared to their method, our algorithm is tailored towards the kind of problems faced in economics. In particular, one of our main contributions is the ability to solve policy functions for which no closed-form solutions are available, an issue very relevant for most standard economic models but not explored in that literature.

Finally, Duarte (2018) is the paper most closely related to ours in that he uses neural network approximators as well as the advantages of the continuous-time formulation. However, his specific algorithm is different from ours. Whilst we perform gradient descent directly on a loss function composed of HJB and first-order condition errors, he employs a method inspired by the reinforcement-learning literature not utilizing the first-order conditions directly. In addition, he chooses his training points uniformly at random instead of from the simulated ergodic distribution. These changes enable us to solve significantly more complex models. As shown below, one of our applications solves a model that has fifty state variables and requires the solution of multiple policy functions for which no closed-form solutions are available. His most

complex application on the other hand includes 10 state variables while also having closed-form solutions available for the single policy function which significantly simplifies the problem.

# 3 Neoclassical Growth Model

Let us now turn to applying our approach to three different setting of increasing complexity. First, we solve a standard continuous-time neoclassical growth model with closed-form solutions for policy functions. This allows us to illustrate our algorithm clearly in a simple setting. Next we compare the solutions of this model to the same model but with a neural network representation of the policy functions. We compare both of these solutions to a reference solution attained with traditional finite difference methods as in Achdou et al. (2020). Finally, we turn to a large scale application that features a 50 continuous states to highlight the strength of our algorithm. Here we consider a social planner who faces a high dimensional dynamic capital allocation problem.

## 3.1 Neoclassical Growth Model with Closed-Form Policy Function

We start with a standard benchmark model, the continuous-time neoclassical growth model. This features closed-form solutions for the policy functions, which allows us to focus our attention on the analysis of the value function approximation. We can then back out the policy function from this approach and compare it to the results of the next step in which we approximate the policy functions themselves with a neural net.

The model takes the canonical form, with a single agent deciding to either save in capital or consume. The HJB equation takes the following form:

$$\rho V(k) = \max_c U(c) + V'(k)[F(k) - \delta * k - c] \tag{4}$$

In this context, one can easily derive an analytical expression for consumption that only depends on the value function and the utility function, namely $c = (U')^{-1}(V'(k))$. With CRRA utility this simplifies further to depend solely on the value function.

Since the model is purely for exhibition of our algorithm we use a standard calibration for all parameters in the model. Utility is CRRA with constant relative risk aversion of $\gamma = 2$. The discount rate $\rho$ is set to 0.04. The production function takes the form $F(k) = A * k^{\alpha}$. Here total factor productivity A is set to 0.5 and $\alpha$, which governs returns to scales, is 0.36. Finally, depreciation is set to 0.05, i.e. a 5 percent annual depreciation rate.

We approximate the value function $V(k)$ with a neural network, $\tilde{V}(k; \Theta)$. Our error criterion is the "HJB error" which is defined as:

$$err_{HJB} = \rho\tilde{V}(k;\Theta) - U\left((U')^{-1}\left(\frac{\partial\tilde{V}(k;\Theta)}{\partial k}\right)\right) - \frac{\partial\tilde{V}(k;\Theta)}{\partial k}\left[F(k) - \delta * k - (U')^{-1}\left(\frac{\partial\tilde{V}(k;\Theta)}{\partial k}\right)\right]$$

The results are shown in the left column of figure 2 below. In particular, panels (a) and (c) contrast the results from our neural network approximation with results obtained through a traditional finite difference scheme as in Achdou et al. (2020). Panel (a) displays the value function, which takes the expected concave shape. Panel (c) displays the consumption policy function, which in this case is directly derived from the value function via the closed-form solutions. Importantly, both value and policy function are virtually identical to the reference solution, displaying the high accuracy of our solution method. This is consistent with the low convergence error shown in panel (e).

## 3.2 Neoclassical Growth Model with Neural Network Approximation for Policy Function

As discussed above our algorithm allows us to solve models that do not have closed-form expressions for the policy functions. In order to test the accuracy of our policy-function approximation, we use the same neoclassical growth model as above. However, this time we do not use the closed-form consumption policy function but rather approximate said policy function directly with a policy neural network $\tilde{C}(k;\Theta^C)$. Adjusting our notation for our value function neural network therefore leads to the following HJB error:

$$err_{HJB} = \rho\tilde{V}(k;\Theta^V) - U\left(\tilde{C}(k;\Theta^C)\right) - \frac{\partial\tilde{V}(k;\Theta^V)}{\partial k}\left[F(k) - \delta * k - \tilde{C}(k;\Theta^C)\right]$$

Similarly we need to include an error that disciplines the policy function for consumption. From the first-order condition for consumption we therefore derive the following error criterion:

$$err_C = (U')^{-1}\left(\frac{\partial\tilde{V}(k;\Theta)}{\partial k}\right) - \tilde{C}(k;\Theta^C)$$

The results for this model are shown in the right column of figure 2. The results in panels (b) and (d) for this approach are virtually identical to both the finite difference solution and by extension to our previous results. As before, panels (f) and (g) display the evolution of both

the HJB and the consumption policy convergence errors.

# 4 Dynamic Capital Allocation Model

In order to evaluate our algorithm and to showcase its strength in large state spaces let us now turn to a multi-location capital allocation model. The model investigates optimal consumption, investment, and cross-location capital flows in a model of aggregate productivity shocks and capital movement frictions. Here, a social planner who maximizes utility over consumption in $L$ locations faces a dynamic capital allocation problem. Each location exhibits idiosyncratic productivity shocks and capital can be moved across locations subject to a capital and investment adjustment cost.

## 4.1 Model Outline

Time is continuous and there are $L$ locations, each with a representative household. The social planner maximizes the sum of the discounted household utilities in each location. Each location $\ell$ has an idiosyncratic and exogenous productivity $z_\ell$, which evolves according to the following diffusion process:

$$dz_\ell = \left[\left(-\nu \log(z_\ell) + \frac{\sigma^2}{2}\right) z_\ell\right] dt + \sigma\, z_\ell dW_t^\ell$$

Households have CRRA utility based on local consumption, governed by the relative risk aversion parameter $\gamma$, $u(c_\ell) = \frac{c_\ell^{1-\gamma}}{1-\gamma}$.

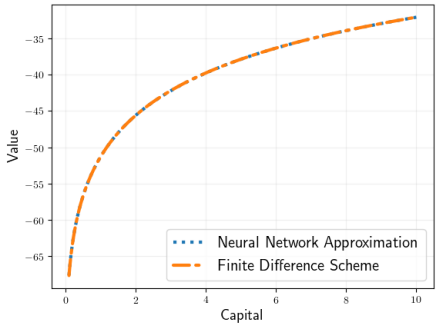
Capital depreciates at rate $\delta$. The social planner can move capital across locations for a quadratic cost. Similarly the social planner faces quadratic investment costs. The overall adjustment costs therefore take the shape:

$$AC(i_\ell, nk_\ell) = \kappa_1 i_\ell^2 + \kappa_2 x_\ell^2$$

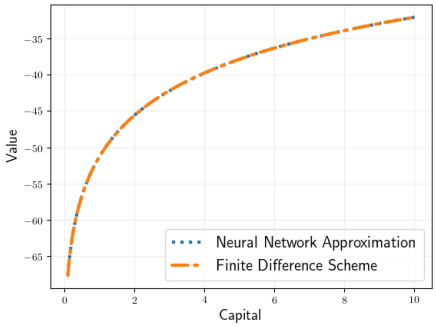
. Output is produced through a standard decreasing returns to scale production function:

$$y_\ell = z_\ell k_\ell^\alpha$$

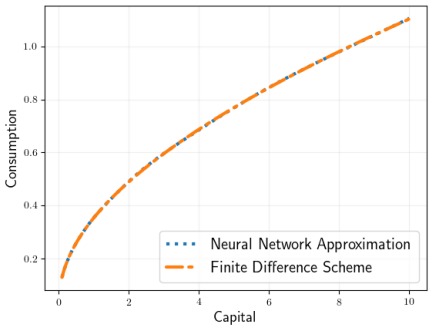
Due to the decreasing returns and consumption smoothing motives, the social planner will move capital between locations.



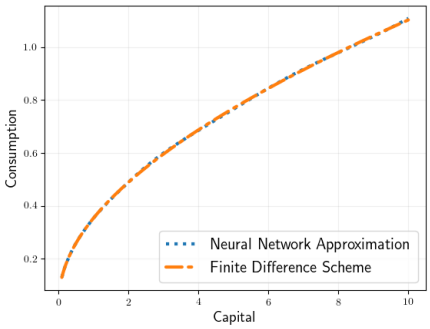

(a) Value with closed-form policy



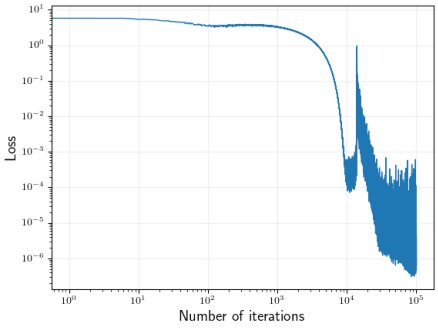

(b) Value with policy approximation



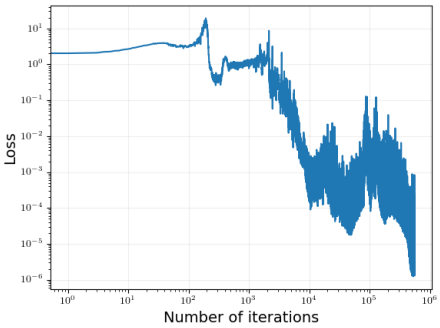

(c) Consumption with closed-form policy



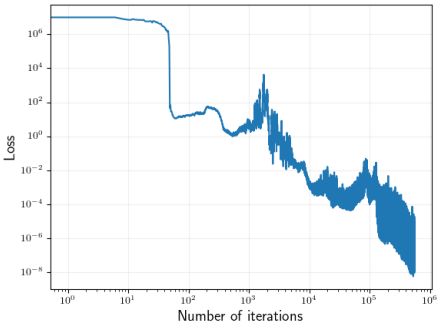

(d) Consumption with policy approximation




(e) HJB error with closed-form policy




(f) HJB error with policy approximation




(g) Policy error with policy approximation

Figure 2: Comparison of results for the Neoclassical growth model with closed-form solutions for policy functions (left column) to the same model with neural net approximations for policy functions (right column). Panels (a) to (d) compare value and policy functions to finite difference solutions, whilst panels (e) - (g) display convergence error evolutions.

The HJB for the social planner's problem is then as follows:

$$\rho V\left(\{k_\ell, z_\ell\}_{\ell=1}^L\right) = \max_{\{c_\ell, i_\ell, x_\ell\}_{\ell=1}^L} \sum_{\ell=1}^L \Bigg[ u(c_\ell) + \partial_{k_\ell} V\left(\{k_\ell, z_\ell\}_{\ell=1}^L\right) \left(i_\ell - \delta k_\ell + x_\ell\right) + \partial_{z_\ell} V\left(\{k_\ell, z_\ell\}_{\ell=1}^L\right) \mu_\ell(z) + \partial_{z_\ell, z_\ell} V\left(\{k_\ell, z_\ell\}_{\ell=1}^L\right) \frac{\sigma_\ell^2}{2} \Bigg]$$

$$\text{s.t.} \begin{cases} c_\ell + i_\ell + AC(i_\ell, x_\ell) = z_\ell k_\ell^\alpha & \forall\, \ell \\ 0 = \sum_{\ell=1}^L x_\ell \end{cases}$$

The flow value of the social planner depends on the instantaneous utility from consumption in each location as well as the evolution of the states. The first constraint represents the budget constraint in each location that the social planner has to observe. Income can either be consumed, invested or shipped to another location, subject to adjustment costs. $x_\ell$ represents the net inflow of capital into location $\ell$. Finally, the second constraint forces the social planner's net capital flows to balance, which is required since the global economy is closed.

## 4.2 Complexity

Since we are dealing with finitely many locations all idiosyncratic shocks are aggregate shocks. Therefore, the social planner needs to keep track of the productivity and capital stock in each location, leading to a state space hyper cube of $2 * L$ dimensions.

Moreover, the social planner has to decide how much capital to invest and how much to export for each location. Based on the constraints this amounts to $L$ consumption choices and $L - 1$ net capital movement policy functions.

## 4.3 Calibration

As this model is mainly for illustrative purposes the calibration is not of central importance. Table 1 gives an overview of the calibration of the parameters. Risk aversion, time preference and returns to scale are set to standard values. Depreciation is set to 7 percent annually. The adjustment costs for investment and capital movements are symmetric and set to 1, purely for illustrative purposes. The process for productivity is adapted from Moll and Itskhoki (2018).

| Parameter | Value | Description |
|---|---|---|
| $\gamma$ | 2.0 | risk-aversion parameter |
| $\rho$ | 0.04 | time preference parameter |
| $\alpha$ | 0.33 | returns to scale |
| $\delta$ | 0.07 | capital depreciation |
| $\kappa_1$ | 1.0 | adjustment costs for investment |
| $\kappa_2$ | 1.0 | adjustment costs for capital movement |
| $e^{-\nu}$ | 0.8 | autocorrelation of productivity |
| $\sigma$ | 0.33 | innovation variance of productivity |

Table 1: Calibration for multi-location capital allocation model

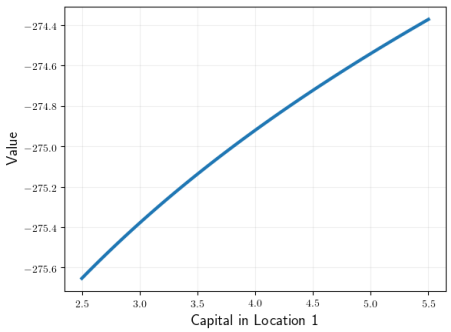
## 4.4 Implementation

We start by defining our neural networks. First, we approximate the value function of the social planner with a neural network, $V\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta^V\right)$. Similarly, we approximate the optimal policy functions through two neural networks, an investment network $i_\ell\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta_\ell^I\right))$ with $L$, and a net capital network $x_\ell\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta_\ell^X\right)$ with $L-1$ outputs. We parameterize these three networks with $\Theta_\ell^V$ $\Theta_\ell^I$, and $\Theta_\ell^X$. Note that we can solve for consumption from the budget constraint.

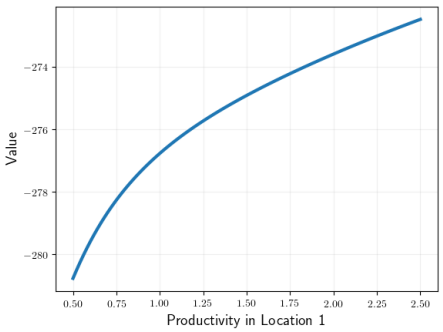
As above let us next define the error criterion on which we train our learning algorithm. We therefore define the individual errors on each of the key equations of our model. The HJB error in the context of our model is given by:

$$\begin{aligned} err_{HJB} = &- \rho\tilde{V}\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta^V\right) + \sum_{\ell=1}^{L}\Bigg[u(c_\ell) + \partial_{k_\ell}\tilde{V}\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta^V\right)\ \dot{k}_\ell \\ &+ \partial_{z_\ell}\tilde{V}\left(\{k_\ell, z_\ell\}_{\ell=1}^L, \Theta^V\right)\ \mu_\ell(z) + \partial_{z_\ell, z_\ell}\tilde{V}\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta^V\right)\ \frac{\sigma_\ell^2}{2}\Bigg] \end{aligned}$$

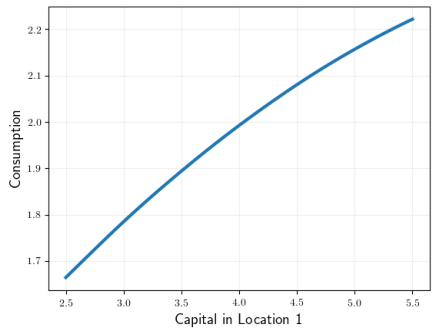
Note that consumption and $\dot{k_\ell}$ depend on the policy choices for investment and net capital movements, for convenience of notation we omitted the dependence here. Next we can turn to the errors that discipline the policy functions. Here we rely on the first-order conditions as well



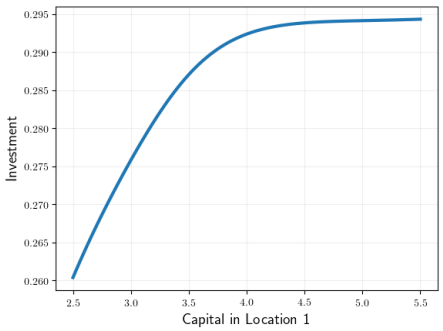

(a) Value over k



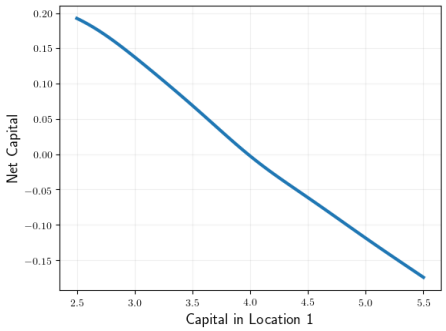

(b) Value over z



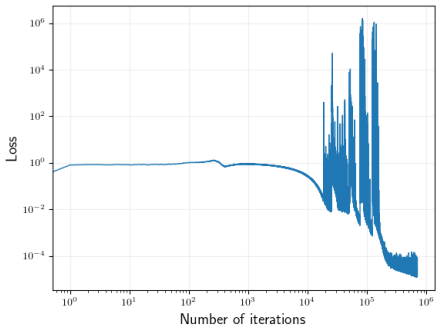

(c) Consumption




(d) Investment




(e) Net Capital




(f) Error

Figure 3: Results for the 25 location capital migration model. We plot partial figures for location 1, with all dimensions not on display fixed at the midpoint of their display intervals.

as the capital market clearing condition.

$$err_{i_\ell} = C_\ell^{-\gamma}(1 + \kappa_1 * i_\ell\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta_\ell^I\right)) - \partial_{k_\ell}\tilde{V}\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta^V\right) \qquad \forall \ell$$

$$err_{x_\ell} = C_\ell^{-\gamma}\kappa_2 x_\ell\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta_\ell^X\right) - \partial_{k_\ell}\tilde{V}\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta^V\right) - C_L^{-\gamma}\kappa_2 x_L\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta_\ell^X\right)$$
$$- \partial_{k_\ell}\tilde{V}\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta^V\right) \qquad \forall \ell \neq L$$

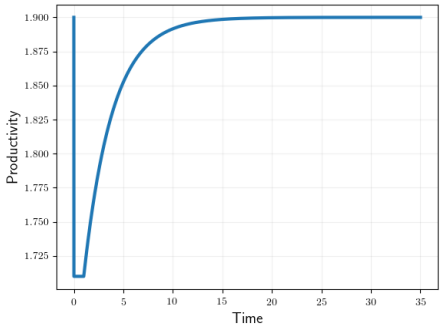
$$err_{clearing} = \sum_{\ell=1}^{L} x_\ell\left(\{k_\ell, z_\ell\}_{\ell=1}^L; \Theta_\ell^X\right)$$

## 4.5 Results

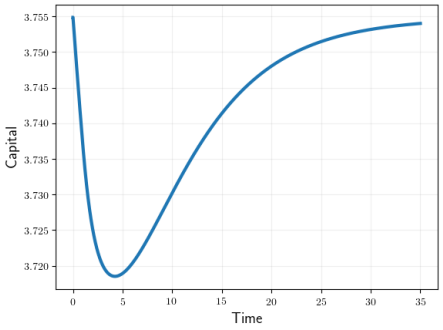
Figure 3 displays the basic results of the model for $L = 25$, which corresponds to 50 continuous state variables, namely capital and productivity for each location. Since the value and all policy functions are 50-dimensional objects, we display partial plots by fixing capital and productivity in all dimensions that are not displayed at the midpoint of their display intervals.

Panel (a) plots the value function in location 1 over capital in location 1, panel (b) displays the same value function over productivity in location 1 and panel (c) shows consumption in location 1. All three curves exhibit the expected concave shape, due to diminishing marginal utility. Investment in panel (d) is also concave, but flattens out for large capital values as the incentive to further accumulate capital decreases with an increase in the capital stock. In comparison to the standard neoclassical growth model, this is further enhanced in this setting by the presence of additional insurance through capital movement from all other locations. Panel (e) displays the net capital movement into location 1. Note that since capital in all other locations is fixed at 4.0, location 1 imports capital for $k < 4.0$, and exports capital for $k > 4.0$ due to the diminishing marginal returns of capital. The exact crossing of the net capital policy function at 4.0 is thus a very reassuring confirmation for our results. In the same vein but more importantly, panel (f) confirms convergence as the overall convergence error decreases to about $10^{-5}$.

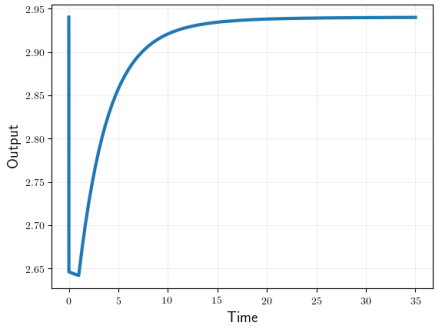
Based on these functions, we can turn to analyzing the dynamics of the model. To do so, let us first compute the deterministic and stochastic steady states. The former is defined as the steady state if there is not uncertainty in the economy, i.e. $\sigma = 0$. On the other hand the stochastic steady state takes uncertainty into account but the realization of the shock happens to always be zero, i.e. $dW = 0$. We derive in the appendix that the respective steady states for productivity in each location are $\bar{z}_{DSS} = 1$ and $\bar{z}_S SS = 1.90$ whilst the steady states for capital are $\bar{k}_{DSS} = 3.04$ and $\bar{k}_{SSS} = 3.75$. This stresses the value of our global solution model, since the underlying uncertainty significantly drives behavior.



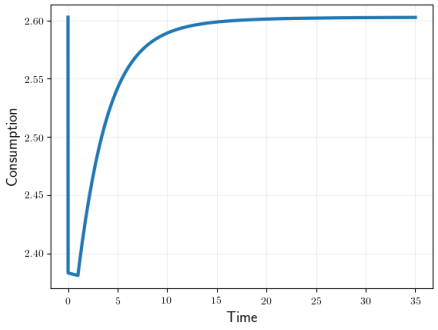

(a) Productivity



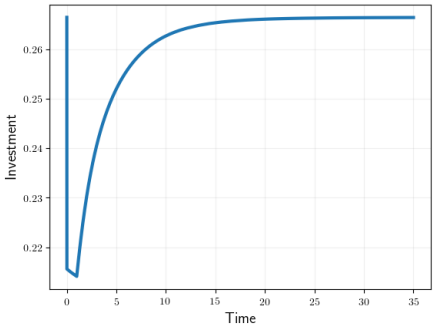

(b) Capital



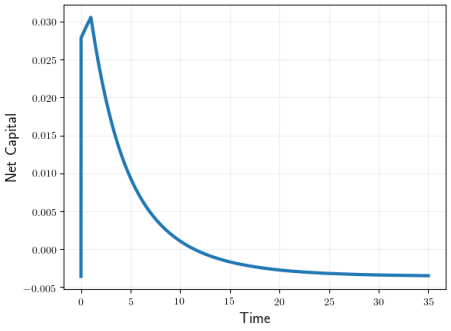

(c) Output




(d) Consumption




(e) Investment




(f) Net Capital

Figure 4: Impulse responses for the 25 location capital migration model to a 10 percent negative productivity shock

To further analyze model behavior, we consider the impulse responses to a 10 percent negative productivity shock in one location. These are displayed in figure 4. Panel (a) displays the evolution of productivity, which drops by 10 percent as the shock hits and then converges back to its steady state with a half-life of two years. Panel (b) shows the response of capital which drops rapidly as the household invests less in order to smooth consumption. After the shock starts to reverse, capital increases back to its steady state, but more slowly as capital accumulation is a gradual process. Panel (c) displays output, which drops in line with productivity and then recovers slightly more slowly due to the slower capital accumulation. Panel (d) displays the drop in consumption, which is significant despite the slowdown in investment and the availability of insurance through capital movement. Panel (e) shows investment which falls as the household smooths consumption. Finally panel (f) displays net capital inflow, which jumps from up as the social planner redistributes capital from other locations to make up for the productivity fall and then falls back to the steady-state value of zero.

# 5 Migration and the Propagation of Shocks

As a final application we study the propagation of economic shocks through a multi-country economy with mobile labor and fully rational expectations. This provides an excellent test for our method, as the model is highly nonlinear and displays kinks in policy functions, illustrating the global solution properties of our method. Additionally, this application is also of great economic interest, as the realistic modeling of migration decisions is understood to be important but has so far eluded the literature due to numerical constraints.

The importance of including migration decisions stems from the fact that migration is strongly driven by economic factors. This has been partially shown in the empirical literature (e.g. Molloy et al., 2011) and we provide further evidence of this in the setting of the 2009 Euro crisis in the appendix. Migration in turn affects a large number of economic variables of interest, so that it is essential to realistically model migration decisions when studying the propagation of shocks.

Due to the inherent complexity of the problem, the previous literature on migration patterns had to contend with simplifying assumptions. In particular, the literature faces four main difficulties. First, it is difficult to model fully rational expectations. For example, Kennan and Walker (2011) restrict agents to only know their economic prospect in one other location. In contrast, our approach allows us to let agents keep track of the entire menu of destinations and their states. This is an important contribution, as Kennan and Walker stress the importance of rational expectations to account for the inherent uncertainty of economic prospects surrounding migration decisions. Second, as argued by Kennan and Walker for a plausible quantification one needs to model a finite number of destinations to be able to speak to the granularity of the data.

Previous migration models with fully rational expectations such as Kaplan and Schulhofer-Wohl (2017) were not able to include a large number of locations due to the curse of dimensionality. Our method is designed for exactly this challenge, and so we are able to include up to 25 different locations, which corresponds to 75 aggregate continuous state variables. Third, as locations are intrinsically heterogeneous, it is important to accommodate locations that are ex ante heterogeneous. Again, our large state space approach enables this without further difficulty. Lastly, migrations are inherently forward-looking choices that are associated with fixed costs of movement. Therefore, it is important to allow for full dynamics in the model. Our model includes this by exhibiting fully forward-looking agents in a dynamic setting.

In conclusion, our approach is the first one able to resolve all four of these concerns, as it allows us to have a fully dynamic rational expectations migration model with a large number of locations. Therefore, to the best of our knowledge, we are the first ones to propose a realistic migration model taking into account all of the above issues.

## 5.1 Model Outline

### 5.1.1 Environment

Time is continuous and there are $L$ locations indexed by $\ell \in \{0, ..., L\}$. In the overall economy there is a mass $L$ of workers that are working in the different countries. In particular, there are $n_\ell$ employed and $u_\ell$ unemployed workers in country $\ell$. The productivity $z_\ell$ in each country follows a country-specific productivity process that develops according to:

$$dz_\ell = \left[\left(-\nu \log(z_\ell) + \frac{\sigma^2}{2}\right) z_\ell\right] dt + \sigma\, z_\ell dW_t^\ell.$$

A country $\ell$ is thus characterized by a triple of state variables $(z_\ell, u_\ell, n_\ell)$ and we summarize the overall state of the economy by $\mathbf{s} = \{z_\ell, u_\ell, n_\ell\}_{\ell=1}^L$.

### 5.1.2 Migration

Households have the option to migrate between the $L$ countries. In particular, with Poisson rate $\alpha$ a household receives the option to migrate. If a household moves from location $\ell$ to location $\ell'$ she incurs welfare costs $\phi + \zeta_{\ell'}$, where $\phi$ is common across all agents, representing fixed migration costs. $\zeta_{\ell'}$ on the other hand is an idiosyncratic migration cost, which we model as an iid random variable distributed according to the distribution $F(\zeta_{\ell'})$. In order to simplify the algebra later on, we assume a logistic distribution $F$ with mean $\mu_F$ and scale $s_F$. This assumption is very similar to the standard assumptions commonly made in the trade literature, for example in Caliendo et al. (2019). If a household decides to move, she arrives at the

new location without a job.[8] We denote the equilibrium rates of movement from location $\ell$ to location $\ell'$ by $p^u_{\ell,\ell'}(\mathbf{s})$ and $p^w_{\ell,\ell'}(\mathbf{s})$ depending on the initial employment state of the moving household.

### 5.1.3 Labor Markets

The labor markets in each country are central to our model, since they determine both the wage rate and the unemployment in the face of migration. We define labor market tightness in country $\ell$, as is conventional in the search literature, as

$$\theta_\ell = \frac{v_\ell}{u_\ell},$$

where $v_\ell$ denotes the vacancies posted in location $\ell$. Given a matching function $m(u_\ell, v_\ell)$, we define the rate of filling a vacancy as

$$q(\theta_\ell) = m\left(\frac{1}{\theta_\ell}, 1\right).$$

We use the commonly employed matching function from Den Haan et al. (2000), that has the convenient property that the matching probabilities are between zero and one:

$$m(u, v) = \frac{uv}{(u+v)^{1/\iota}},$$

where $\iota > 0$ is the matching elasticity. In addition to endogenous job destruction due to migration, jobs are also destroyed exogenously at rate $\delta$. Finally, wages are set according to Nash Bargaining, which we will discuss in more detail below.

### 5.1.4 Household Problem

Households living in this economy discount the future at rate $\rho$ and receive instantaneous CRRA utility $U(c) = \frac{c^{1-\gamma}}{1-\gamma}$ from consumption $c$. In case a household is unemployed, she generates income $b$ from home production.

Based on the above setup, unemployment in country $\ell$ evolves according to the following law of motion:

$$\dot{u}_\ell(\mathbf{s}) = \delta_\ell n_\ell + \sum_{\ell' \neq \ell} \left( p^w_{\ell',\ell}(\mathbf{s}) n_{\ell'} + p^u_{\ell',\ell}(\mathbf{s}) u_{\ell'} \right) - \left( \sum_{\ell' \neq \ell} p^u_{\ell,\ell'}(\mathbf{s}) + q(\theta_\ell)\theta_\ell \right) u_\ell.$$

[8] Note that different modeling choices are possible here, e.g. households could also arrive in their destination country with a new job or alternatively with some probability of a new job. We opt for our approach since it simplifies the problem and in the data most migrating households engage in a job search after arrival in the target country.

Similarly, employment in country $\ell$ evolves according to:

$$\dot{n}_\ell(\mathbf{s}) = q(\theta_\ell)v_\ell - \left(\sum_{\ell' \neq \ell} p^w_{\ell,\ell'}(\mathbf{s}) + \delta_\ell\right) n_\ell.$$

The recursive problems of the unemployed and the employed households summarize the household problem. In particular, denoting by $V^u(\hat{\ell};\mathbf{s})$ the value function of the unemployed household in location $\hat{\ell}$ with the economy being in state $\mathbf{s}$, the HJB of an unemployed household is given by:

$$\begin{aligned}\rho V^u(\hat{\ell};\mathbf{s}) =& U(b) + \underbrace{\sum_{\ell=1}^{L}\left[V^u_{z_\ell}(\hat{\ell};\mathbf{s})\mu(z_\ell) + \frac{\sigma(z_\ell)^2}{2}V^u_{z_\ell z_\ell}(\hat{\ell};\mathbf{s})\right]}_{\text{evolution of productivities}} + \underbrace{\sum_{\ell=1}^{L}\left[V^u_{u_\ell}(\hat{\ell};\mathbf{s})\dot{u}_\ell(\mathbf{s}) + V^u_{n_\ell}(\hat{\ell};\mathbf{s})\dot{n}_\ell(\mathbf{s})\right]}_{\text{evolution of (un)employed population}} \\ &+ \underbrace{q(\theta_\ell)\theta_\ell\Big(V^u(\hat{\ell};\mathbf{s}) - V^w(\hat{\ell};\mathbf{s})\Big)}_{\text{job finding}} + \underbrace{\alpha\sum_{\ell'\neq\hat{\ell}}\int_{-\infty}^{\bar{\zeta}^u_{\hat{\ell},\ell'}(\mathbf{s})}\left[V^u(\ell';\mathbf{s}) - V^u(\hat{\ell};\mathbf{s}) - \phi_{\hat{\ell},\ell'} - \zeta_{\ell'}\right]dF(\zeta_{\ell'})}_{\text{migration decision}}\end{aligned}$$

$$\begin{aligned}&\text{s.t. } \bar{\zeta}^u_{\hat{\ell},\ell'}(\mathbf{s}) = V^u(\ell';\mathbf{s}) - V^u(\hat{\ell};\mathbf{s}) - \phi_{\hat{\ell},\ell'}, \\ &\quad \dot{u} \text{ and } \dot{n} \text{ follow their laws of motion}\end{aligned}$$

The equivalent problem for the employed household with $V^w(\hat{\ell};\mathbf{s})$ denoting her value function is:

$$\begin{aligned}\rho V^w(\hat{\ell};\mathbf{s}) =& U(w(\hat{\ell};\mathbf{s})) + \underbrace{\sum_{\ell=1}^{L}\left[V^w_{z_\ell}(\hat{\ell};\mathbf{s})\mu(z_\ell) + \frac{\sigma(z_\ell)^2}{2}V^w_{z_\ell z_\ell}(\hat{\ell};\mathbf{s})\right]}_{\text{evolution of productivities}} + \underbrace{\sum_{\ell=1}^{L}\left[V^w_{u_\ell}(\hat{\ell};\mathbf{s})\dot{u}_\ell(\mathbf{s}) + V^w_{n_\ell}(\hat{\ell};\mathbf{s})\dot{n}_\ell(\mathbf{s})\right]}_{\text{evolution of (un)employed population}} \\ &+ \underbrace{\delta_{\hat{\ell}}\Big(V^u(\hat{\ell};\mathbf{s}) - V^w(\hat{\ell};\mathbf{s})\Big)}_{\text{job loss}} + \underbrace{\alpha\sum_{\ell'\neq\hat{\ell}}\int_{-\infty}^{\bar{\zeta}^w_{\hat{\ell},\ell'}(\mathbf{s})}\left[V^u(\ell';\mathbf{s}) - V^w(\hat{\ell};\mathbf{s}) - \phi_{\hat{\ell},\ell'} - \zeta_{\ell'}\right]dF(\zeta_{\ell'})}_{\text{migration decision}}\end{aligned}$$

$$\begin{aligned}&\text{s.t. } \bar{\zeta}^w_{\hat{\ell},\ell'}(\mathbf{s}) = V^u(\ell';\mathbf{s}) - V^w(\hat{\ell};\mathbf{s}) - \phi_{\hat{\ell},\ell'}, \\ &\dot{u} \text{ and } \dot{n} \text{ follow their laws of motion}\end{aligned}$$

### 5.1.5 Firm Problem

In each country there is a forward-looking atomistic representative firm that posts vacancies $v_\ell$ subject to vacancy-posting costs $\kappa(\theta_\ell)$, where $\kappa(\theta_\ell) = \kappa_0 + \kappa_1 q(\theta_\ell)$ as in Pissarides (2009).[9]

[9]Here $\kappa_1 > 0$ represents the fixed costs of hiring, whilst $\kappa_0/q(\theta_\ell)$ are the marginal costs of hiring which are increasing in the mean duration of vacancies $1/q(\theta_\ell)$. We implement these convex vacancy-posting costs in order

Since firms are owned by households we assume that they discount the future also at rate $\rho$.[10] The firm produces according to a constant returns to scale production function taking labor as its sole input $f(z_\ell, n_\ell) = z_\ell n_\ell$. Denoting by $\tilde{n}_\ell$ the employment of the firm, its law of motion is given by

$$\dot{\tilde{n}}_\ell(\mathbf{s}) = q(\theta_\ell)\tilde{v} - \left(\sum_{\ell' \neq \ell} p^w_{\ell,\ell'}(\mathbf{s}) + \delta_\ell\right)\tilde{n}_\ell$$

The firm problem is then summarized by its HJB equation:

$$\rho J(\hat{\ell}, \tilde{n}; \mathbf{s}) = \max_{\tilde{v}\geq 0} z_{\hat{\ell}}\tilde{n}_{\hat{\ell}} - w_{\hat{\ell}}(\hat{\ell}; \mathbf{s})\tilde{n}_{\hat{\ell}} - \kappa\left(\theta_{\hat{\ell}}\right)\tilde{v} + \underbrace{\sum_{\ell=1}^{L}\left[J_{z_\ell}(\hat{\ell}, \tilde{n}; \mathbf{s})\mu(z_\ell) + \frac{\sigma(z_\ell)^2}{2}J_{z_\ell z_\ell}(\hat{\ell}, \tilde{n}; \mathbf{s})\right]}_{\text{evolution of productivities}} +$$

$$+ \underbrace{\sum_{\ell=1}^{L}\left[J_{u_\ell}(\hat{\ell}, \tilde{n}; \mathbf{s})\dot{u}_\ell(\mathbf{s}) + J_{n_\ell}(\hat{\ell}, \tilde{n}; \mathbf{s})\dot{n}_\ell(\mathbf{s})\right]}_{\text{evolution of (un)employed population}} + \underbrace{J_{\tilde{n}_{\hat{\ell}}}(\hat{\ell}, \tilde{n}; \mathbf{s})\dot{\tilde{n}}_{\hat{\ell}}(\mathbf{s})}_{\text{evolution of firm employment}}$$

### 5.1.6 Equilibrium

The Recursive Competitive Equilibrium of this economy is defined by value and policy functions $\{\tilde{v}_\ell, \tilde{n}_\ell, V^w(\ell), V^u(\ell), J(\ell), \bar{\zeta}^w_{\ell,\ell'}, \bar{\zeta}^u_{\ell,\ell'}\}_{\ell=1}^L$, wages $w(\ell)$, and moving rates $p^w_{\ell,\ell'}$ and $p^u_{\ell,\ell'}$ over the state vector $\mathbf{s} = \{z_\ell, u_\ell, n_\ell\}_{\ell=1}^L$ such that:

1. Given $\{w(\ell; \mathbf{s}), p^w_{\ell,\ell'}, p^u_{\ell,\ell'}\}$ for all $\ell, \ell' \in 1, ..., L$ : $V^w, V^u$ and $J$ solve the employed household HJB, the unemployed household HJB and the firm HJB.

2. Labor market clear: $n_\ell = \tilde{n}_\ell$ for all $\ell$.

3. Firm and aggregate vacancies are consistent: $v_\ell = \tilde{v}_\ell$ for all $\ell$.

4. The moving rates are given by: $p^w_{\ell,\ell'}(\mathbf{s}) = \alpha F(\bar{\zeta}^w_{\ell,\ell'}(\mathbf{s}))$ and $p^u_{\ell,\ell'}(\mathbf{s}) = \alpha F(\bar{\zeta}^u_{\ell,\ell'}(\mathbf{s}))$.

5. The wage rate $w(\ell; \mathbf{s})$ is consistent with Nash Bargaining.

6. The laws of motion for $\mathbf{s}$ hold.

to limit firm size in the absence of decreasing returns to scale.

[10] Formally, the firm discounts the future at the stochastic discount factor of its owners. Since the firm is fully owned by households and we have representative households without idiosyncratic states, this stochastic discount factor is equal to $\rho$.

### 5.1.7 Equilibrium Implications

We can take first-order conditions in the firm problem, which yields

$$\kappa\left(\theta_{\hat{\ell}}\right) = q(\theta_{\hat{\ell}}) J_{n_{\hat{\ell}}}(\hat{\ell}; \mathbf{s}).$$

Plugging in our functional form for $\kappa(\theta_\ell)$, this is equivalent to

$$\frac{\kappa_0}{q(\theta_{\hat{\ell}})} + \kappa_1 = J_{n_{\hat{\ell}}}(\hat{\ell}; \mathbf{s}),$$

which implicitly defines $\theta_{\hat{\ell}} = \theta(\hat{\ell}; \mathbf{s})$. Turning to the determination of wages, we can express the standard Nash Bargaining condition in this context as:

$$w_{\hat{\ell}} = \arg\max_w \left(V^w(\hat{\ell}; \mathbf{s}) - V^u(\hat{\ell}; \mathbf{s})\right)^\beta \left(J(\hat{\ell}; \mathbf{s})\right)^{1-\beta},$$

which yields the following first-order condition:

$$V^w(\hat{\ell}; \mathbf{s}) - V^u(\hat{\ell}; \mathbf{s}) = \frac{\beta}{1-\beta} J(\hat{\ell}; \mathbf{s}).$$

Finally, due to $n_\ell = \tilde{n}_\ell$ for all $\ell$ in equilibrium, we have $J(\ell, \tilde{n}; \mathbf{s}) = J(\ell, \mathbf{s})$, so that the last term in the firm HJB drops out.

## 5.2 Complexity

As above, since we are dealing with a finite set of locations, all idiosyncratic shocks are aggregate shocks. In addition to their own states, agents with fully rational expectations need to keep track of the productivities as well as the employment and unemployment states in all other locations in order to make fully-informed and optimal migration decisions. Therefore, the state space has $3 \times L$ dimensions. Furthermore, this model is highly nonlinear, featuring kinks as we will show below. It is thus not only an interesting economic application but also a great showcase for the global solution capabilities of our method.

## 5.3 Calibration

Besides showcasing the power of our algorithm, the purpose of this model is to study the effects of migration on the propagation of shocks in the presence of multiple countries. Since this application is general instead of being focused on one particular set of countries, we aim for a generally reasonable calibration instead of a country-specific one. Table 2 displays our calibrated values for the parameters in our model. We only discuss those parameters that are

| Parameter | Value | Description |
|---|---|---|
| $\gamma$ | 2.0 | risk-aversion parameter |
| $\rho$ | 0.04 | time preference parameter |
| $\delta$ | 0.04 | exogenous job destruction |
| $e^{-\nu}$ | 0.8 | autocorrelation of productivity |
| $\sigma$ | 0.33 | innovation variance of productivity |
| $\mu_F$ | 0.6 | mean of migration cost distribution |
| $s_F$ | 0.6 | std of migration cost distribution |
| $\alpha$ | 0.1 | migration opportunity shock |
| $\phi$ | 0.9 | common migration cost |
| $\iota$ | 1.25 | elasticity of matching function |
| $\beta$ | 0.04 | Nash bargaining power of workers |
| $b$ | 0.31 | Home Production |
| $\kappa_0$ | 0.5 | Vacancy posting costs 1 |
| $\kappa_1$ | 0.5 | Vacancy posting costs 2 |

Table 2: Calibration for multi-location capital allocation model

different from the ones discussed in the Dynamic Capital Allocation Model discussed above. We set the exoegnous job destruction rate $\delta$ to 4% in the range of values estimated by Davis et al. (2006). The mean of the logistic movement cost distribution $\mu_F$ is 0.6 and its scale $s_F$ 1, to match the moving cost distribution in Kennan and Walker (2011). We set the common migration cost $\phi$ to 0.9 to be 50% higher than the mean idiosyncratic migration shock. The elasticity of the matching function $\iota$ is 1.25 as in Den Haan et al. (2000). The Nash bargaining weight of workers is set to 0.04 as in Petrosky-Nadeau et al. (2018). Regarding home production we set $b$ in line with the labor wedge literature. In particular, we choose an intermediate value of 0.4, which corresponds to 0.85 ones one normalizes for mean productivity. We choose this value since it is a reasonable value commonly used, that is between the extreme values 0.4 of Shimer (2005) and 0.96 in Hagedorn and Manovskii (2008). Finally, we choose $\kappa_0$ and $\kappa_1$ to be equal to 0.5 as in Petrosky-Nadeau et al. (2018).

## 5.4 Implementation

We have now assembled all the ingredients to solve the model. First, note that thanks to the assumption of a logistic distribution for the idiosyncratic migration cost, we can derive

closed-form solutions for the two integrals in the employed and unemployed household HJBs:

$$\int_{-\infty}^{\bar{\zeta}^u_{\hat{\ell},\ell'}(\mathbf{s})} \left[V^u(\ell';\mathbf{s}) - V^u(\hat{\ell};\mathbf{s}) - \phi_{\hat{\ell},\ell'} - \zeta_{\ell'}\right] dF\left(\zeta_{\ell'}\right) = \frac{V^u(\ell';\mathbf{s}) - V^u(\hat{\ell};\mathbf{s}) - \phi_{\hat{\ell},\ell'}}{1 + e^{-\frac{\left(\bar{\zeta}^u_{\hat{\ell},\ell'}(\mathbf{s}) - \mu_F\right)}{s_F}}} - \left(\frac{\bar{\zeta}^u_{\hat{\ell},\ell'}(\mathbf{s}) e^{\bar{\zeta}^u_{\hat{\ell},\ell'}(\mathbf{s})/s_F}}{e^{\mu_F/s_F} + e^{\bar{\zeta}^u_{\hat{\ell},\ell'}(\mathbf{s})/s_F}} + \log\left(\frac{e^{\mu_F}}{e^{\mu_F} + e^{\bar{\zeta}^u_{\hat{\ell},\ell'}(\mathbf{s})}}\right)\right),$$

$$\int_{-\infty}^{\bar{\zeta}^w_{\hat{\ell},\ell'}(\mathbf{s})} \left[V^u(\ell';\mathbf{s}) - V^w(\hat{\ell};\mathbf{s}) - \phi_{\hat{\ell},\ell'} - \zeta_{\ell'}\right] dF\left(\zeta_{\ell'}\right) = \frac{V^u(\ell';\mathbf{s}) - V^w(\hat{\ell};\mathbf{s}) - \phi_{\hat{\ell},\ell'}}{1 + e^{-\frac{\left(\bar{\zeta}^w_{\hat{\ell},\ell'}(\mathbf{s}) - \mu_F\right)}{s_F}}} - \left(\frac{\bar{\zeta}^w_{\hat{\ell},\ell'}(\mathbf{s}) e^{\bar{\zeta}^w_{\hat{\ell},\ell'}(\mathbf{s})/s_F}}{e^{\mu_F/s_F} + e^{\bar{\zeta}^w_{\hat{\ell},\ell'}(\mathbf{s})/s_F}} + \log\left(\frac{e^{\mu_F}}{e^{\mu_F} + e^{\bar{\zeta}^w_{\hat{\ell},\ell'}(\mathbf{s})}}\right)\right).$$

This is crucial, since otherwise we would face an integral approximation problem. As discussed in the Solution section above this is a tremendously difficult problem in high dimensions, which our algorithm elegantly circumvents.

To implement the algorithm we now only need to determine which objects to approximate with neural networks and define an appropriate loss function.

The first step is straightforward. Since we do not have any policy functions, for which we do not have closed-form solutions, we only need to represent the three value functions $J(\ell;\mathbf{s})$, $V^w(\ell;\mathbf{s})$ and $V^u(\ell;\mathbf{s})$ with neural networks $\tilde{J}(\ell,\mathbf{s};\Theta^J_\ell)$, $\tilde{V}^w(\ell,\mathbf{s};\Theta^W_\ell)$ and $\tilde{V}^u(\ell,\mathbf{s};\Theta^U_\ell)$.

The definition of the loss function requires a little more work. First, we use the firm's first-order condition

$$\frac{\kappa_0}{q(\theta_\ell)} + \kappa_1 = \tilde{J}_{n_\ell}(\ell,\mathbf{s};\Theta^J_\ell)$$

to compute $\theta_\ell(\mathbf{s})$, employing $\tilde{J}$. Second, we can invert the HJB of the employed household to derive a closed-form solution for $w(\ell;\mathbf{s})$:

$$w(\ell;\mathbf{s}) = U^{-1}[\rho\tilde{V}^w(\ell,\mathbf{s};\Theta^W_\ell - R)],$$

where $R$ is equal to the entire right-hand side of the employed household's value function with the exception of $U(\cdot)$. Note that everything in this remainder depends only on either $\theta_\ell$ or the three neural networks and can thus be computed efficiently.

Finally, we can define three error criteria that together form the loss function. In particular, we employ the HJB error of the unemployed household $HJB^u(\ell,\mathbf{s};\Theta^W_\ell,\Theta^U_\ell,\Theta^J_\ell)$, i.e.

the difference between its left and right-hand side, and equivalently the HJB error of the firm $HJB^J(\ell, \mathbf{s}; \Theta^W_\ell, \Theta^U_\ell, \Theta^J_\ell)$. The final error term comes from the Nash Bargaining Condition. We can thus define the overall loss function as follows:

$$\mathcal{L}(\mathbf{s}; \Theta^W, \Theta^U, \Theta^J) = \sum_{\ell=1}^{L} \Big[ ||HJB^u(\ell, \mathbf{s}; \Theta^W_\ell, \Theta^U_\ell, \Theta^J_\ell)||_2 + ||HJB^J(\ell, \mathbf{s}; \Theta^W_\ell, \Theta^U_\ell, \Theta^J_\ell)||_2 + \Big|\Big| \tilde{V}^w(\ell, \mathbf{s}; \Theta^W_\ell) - \tilde{V}^u(\ell, \mathbf{s}; \Theta^V_\ell) - \frac{\beta}{1-\beta} \tilde{J}(\ell, \mathbf{s}; \Theta^J_\ell) \Big|\Big|_2 \Big]$$

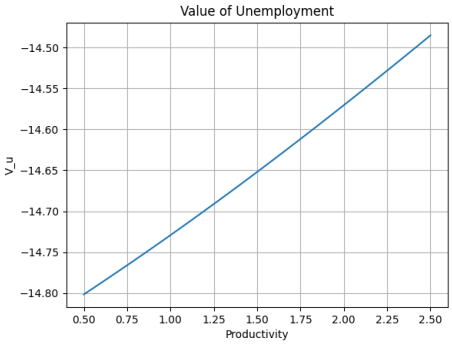
## 5.5 Results

Figure 5 displays the results of this model for $L = 25$, which corresponds to 75 continuous aggregate state variables. With this number of location, this model is significantly more complex than comparable papers in the literature, e.g. Kaplan and Schulhofer-Wohl (2017) who only use two locations. Therefore, in order to solve it, our solution method is required as the complexity is beyond the scope of other methods.

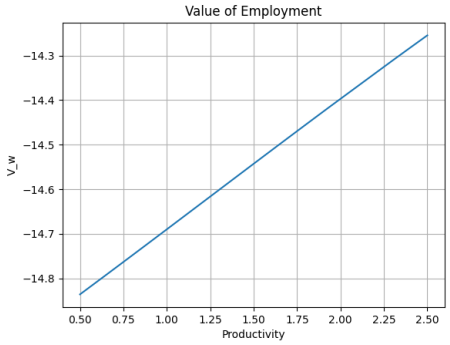
Panel (a) shows the unemployed household's value function, panel (b) the employed household's value function and panel (c) the firm's value function. These all take the expected shape, namely being close to linear and increasing in productivity. This is the result of the constant returns to scale production function in the firm's problem. This implies that the marginal value of an extra worker to the firm is equal to productivity, which extends to the households' value functions through Nash Bargaining.

The most interesting graph, however, is panel (d), which displays labor market tightness $\theta$ plotted over productivity. Note that since $u$ is fixed at its steady-state value, this plot is equivalent to the number of vacancies posted $v$. The number of vacancies is equal to zero up to a threshold value and is then increasing and concave. Again, this is the expected shape and its underlying logic is intuitive. Given the fixed costs of vacancy creation, firms do not post vacancies for low values of productivity. Once the threshold is crossed, they start posting an increasing number of vacancies. Due to the increasing marginal costs of posting a vacancy this increase is not linear in productivity but instead concave. Critically, this plot displays the inherent nonlinearity in the model. In particular, we have a kink in this essential policy function. For this reason, this model can only be solved with a global solution method, demonstrating the power of our approach to solve very large state-space models globally.

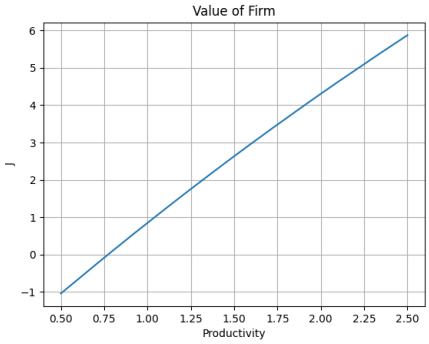
Finally, panel (e) displays the evolution of the loss function. Notice that it drops to below $10^{-1}$. Analyzing this error, we first note that it is mechanically increasing in the number of locations, since we add more non-negative HJB and Nash Bargaining errors as the number of locations increases. Once we look at the individual HJB errors, they drop to approximately $10^{-4}$, which amounts to a remarkable precision given the complexity of the problem.



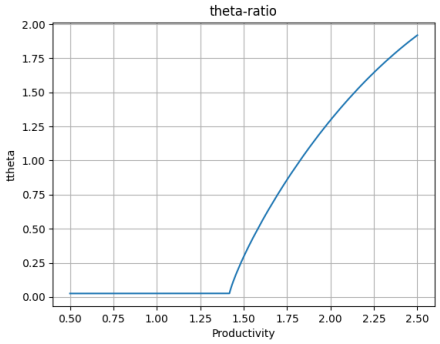

(a) Unemployed household value function



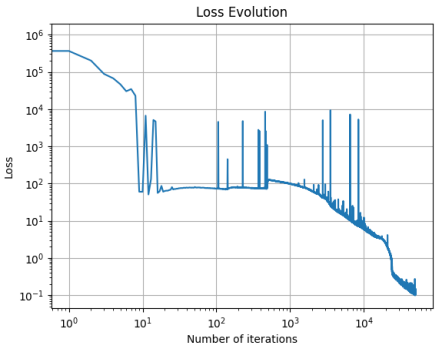

(b) Employed household value function




(c) Firm Value Function




(d) Market Tightness




(e) Loss Function

Figure 5: Results of the multi-country migration model with $L = 25$, corresponding to 75 continuous aggregate state variables. The three value functions and market tightness are plotted over productivity in location 1, with the other states being fixed at their steady-state values.

Overall, these results thus demonstrate that our model is capable of solving a highly nonlinear model with a very large state space with high precision. Whilst we mostly use this model as an illustration of the power of our method here, in future work we will investigate the underlying economics more closely.

# 6 Conclusion

In this paper we propose a novel approach for globally solving large state space models. In order to overcome the curse of dimensionality we rely on neural networks to approximate both value and policy functions. By utilizing the advantages of a continuous-time formulation, our algorithm sidesteps the computationally intensive and potentially inaccurate approximation of expectations over the large state space hypercube.

Our algorithm draws on the HJB to discipline the value function, while using the FOCs and additional constraints as convergence errors for the policy functions. This approach is feasible since the otherwise expensive calculation of the various derivatives in these errors is provided efficiently in neural network implementations.

Finally, our algorithm draws on an approximation of the ergodic distribution based on the policy functions to further curb the curse of dimensionality. By focusing only on the manifold in the state space hypercube that is part of the support of the ergodic distribution we can sharpen our approximation of the value and policy functions in the relevant area.

Next we apply our algorithm to three different settings. First we solve the standard neoclassical growth model and compare our results to those of a reference approach using a finite difference scheme. This allows us to investigate the accuracy of our general approach and show that our policy function approximation works well. Next, in order to demonstrate the ability of our approach to handle large scale problems we apply our algorithm to a dynamic model of capital allocation. Here we solve a model with 50 continuous state variables that requires the approximation of 50 policy functions in addition to the value function. Finally, we solve a large scale migration model with 75 continuous aggregate state variables. The inherent non-linearity of this model showcases the global properties of our method. This illustrates the power of our method as this class of complexity was previously unsolvable.

Based on our success in this context our method lends itself to a variety of future applications. Specifically, there is a broad class of models that our algorithm addresses, namely models concerned with granular data and frictions. We refer to these models as "Node" models, where a node is a distinct unit of economic interest, such as an agent, a region, an industry or a firm. Importantly, as long as there are finitely many nodes, shocks to an individual node do not wash out in the aggregate. Consequently, each agent has to keep track of the state of each node in her forward-looking decision problem. This gives rise to large state spaces for which our method is

well-suited. We therefore see our methodology as an avenue to address questions such as how do productivity shocks propagate through production networks or how links between financial institutions can amplify shocks through a financial network.

In future work we will examine the propagation of shocks in a multi-country model with migration more closely. In particular, whilst in the present paper we mainly use the above migration model as an illustration of the power of our solution method, the model is also of inherent economic interest. Specifically, we are interested in understanding what implications a free movement zone like Schengen has on the propagation of shocks across member countries. In addition, we aim to investigate to what extend the size and composition of such a zone matter. With this applications and the general class of node models identified above as examples, we are convinced that our new solution method enables future research linking macroeconomic models and granular data.

# References


Azinović, M., L. Gaegauf, and S. Scheidegger (2019): "Deep Equilibrium Nets," Mimeo, University of Zurich, https://ssrn.com/abstract=3393482.

Bach, F. (2017): "Breaking the curse of dimensionality with convex neural networks," *Journal of Machine Learning Research*, 18, 629–681.

Barron, A. (1993): "Universal approximation bounds for superpositions of a sigmoidal function," *IEEE Transactions on Information Theory*, 39, 930–945.

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind (2018): "Automatic differentiation in machine learning: a survey," *The Journal of Machine Learning Research*, 18, 1–43.

Bellman, R. (1958): *Dynamic programming*, Princeton University Press.

Boyd, J. P. (2001): *Chebyshev and Fourier spectral methods*, Mineola, N.Y: Dover Publications, 2nd ed., rev ed.

Boyd, J. P. and R. Petschek (2014): "The Relationships Between Chebyshev, Legendre and Jacobi Polynomials: The Generic Superiority of Chebyshev Polynomials and Three Important Exceptions," *Journal of Scientific Computing*, 59, 1–27.

Brumm, J. and S. Scheidegger (2017): "Using Adaptive Sparse Grids to Solve HighDimensional Dynamic Models," *Econometrica*, 85, 1575–1612.

Caliendo, L., M. Dvorkin, and F. Parro (2019): "Trade and Labor Market Dynamics: General Equilibrium Analysis of the China Trade Shock," *Econometrica*, 87, 741–835.

Cover, T. and J. Thomas (2012): *Elements of Information Theory, 2nd edition*, Wiley.

Cybenko, G. (1989): "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems*, 2, 303–314.

Davis, S. J., R. J. Faberman, and J. Haltiwanger (2006): "The Flow Approach to Labor Markets: New Data Sources and MicroMacro Links," *Journal of Economic Perspectives*, 20, 3–26.

Den Haan, W. J., G. Ramey, and J. Watson (2000): "Job Destruction and Propagation of Shocks," *American Economic Review*, 90, 482–498.

Duarte, V. (2018): "Machine Learning for Continuous-Time Finance," .

Fernández-Villaverde, J. and P. A. Guerrón-Quintana (2020): "Estimating DSGE Models: Recent Advances and Future Challenges," Working Paper 27715, National Bureau of Economic Research.

Fernández-Villaverde, J., S. Hurtado, and G. Nuño (2019): "Financial Frictions and the Wealth Distribution," Working Paper 26302, National Bureau of Economic Research.

Glorot, X. and Y. Bengio (2010): "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Aritficial Intelligence and Statistics, JMLR Workshop and Conference Proceedings*, vol. 9, 249–256.

Goodfellow, I., Y. Bengio, and A. Courville (2016): *Deep Learning*, MIT Press.

Hagedorn, M. and I. Manovskii (2008): "The Cyclical Behavior of Equilibrium Unemployment and Vacancies Revisited," *American Economic Review*, 98, 1692–1706.

Han, J., A. Jentzen, and W. E (2018): "Solving high-dimensional partial differential equations using deep learning," *Proceedings of the National Academy of Sciences*, 115, 8505–8510.

Hornik, K., M. Stinchcombe, and H. White (1989): "Multilayer feedforward networks are universal approximators," *Neural Networks*, 2, 359–366.

Kaplan, G. and S. Schulhofer-Wohl (2017): "Understanding the Long-Run Decline in Interstate Migration," *International Economic Review*, 58, 57–94.

Kennan, J. and J. R. Walker (2011): "The Effect of Expected Income on Individual Migration Decisions," *Econometrica*, 79, 211–251.

Maliar, L., S. Maliar, and P. Winant (2019): "Will Artificial Intelligence Replace Computational Economists Any Time Soon?" CEPR Discussion Papers 14024, C.E.P.R. Discussion Papers.

Molloy, R., C. L. Smith, and A. Wozniak (2011): "Internal Migration in the United States," *Journal of Economic Perspectives*, 25, 173–196.

Petrosky-Nadeau, N., L. Zhang, and L.-A. Kuehn (2018): "Endogenous Disasters," *American Economic Review*, 108, 2212–2245.

Pissarides, C. (2009): "The Unemployment Volatility Puzzle: Is Wage Stickiness the Answer?" *Econometrica*, 77, 1339–1369.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986): "Learning representations by back-propagating errors," *Nature*, 323, 533–536.

Shimer, R. (2005): "The Cyclical Behavior of Equilibrium Unemployment and Vacancies," *American Economic Review*, 95, 25–49.

Sirignano, J. and K. Spiliopoulos (2018): "DGM: A deep learning algorithm for solving partial differential equations," *Journal of Computational Physics*, 375, 1339–1364.

Sutton, R. S. and A. G. Barto (2018): *Reinforcement Learning: An Introduction*, Cambridge, MA, USA: A Bradford Book.

# 7 Appendix

## 7.1 Proof for continuous HJB

Here we provide a heuristic proof of the HJB equation which highlights the link with the discrete time formulation.

We thus show that the discrete-time problem (3) is equal to the continuous-time HJB (1) as $dt \to 0$. We start with the discrete-time Bellman equation:

$$V(\mathbf{x}_t) = \max_{\alpha} r(\mathbf{x}_t, \alpha) + \beta \mathbb{E}\left[V(\mathbf{x}_{t+1})\right].$$

If we express it in continuous time over a fixed interval of size $\Delta t$ and define $\rho \equiv -\frac{1}{\Delta t}\log\beta$, we get

$$V(\mathbf{x}_t) = \max_{\{\alpha_{t+s}\}_{s\in[0,\Delta t]}} \int_0^{\Delta t} r(\mathbf{x}_{t+s}, \alpha_{t+s})\, ds + e^{-\rho\Delta t}\mathbb{E}\left[V(\mathbf{x}_{t+\Delta t})\right].$$

Given the optimal policy $\alpha^*$, if we take the derivative with respect to $\Delta t$, we obtain

$$0 = r\left(\mathbf{x}_{t+\Delta t}, \alpha^*_{t+\Delta t}\right) - \rho e^{-\rho\Delta t}\mathbb{E}\left[V(\mathbf{x}_{t+\Delta t})\right] + e^{-\rho\Delta t}\mathbb{E}\left[\frac{dV(\mathbf{x}_{t+\Delta t})}{dt}\right].$$

Applying It's lemma and taking expectations,

$$\begin{aligned}
\frac{1}{dt}\mathbb{E}\left[dV(\mathbf{x}_{t+\Delta t})\right] &= \frac{1}{dt}\mathbb{E}\Big[\left(\nabla_x V(\mathbf{x}_{t+\Delta t})\right)^T f\left(\mathbf{x}_{t+\Delta t}, \alpha^*_{t+\Delta t}\right) \\
&+ \frac{1}{2}Tr\left[\left(\sigma(\mathbf{x}_{t+\Delta t})\right)^T \Delta_x V(\mathbf{x}_{t+\Delta t})\,\sigma(\mathbf{x}_{t+\Delta t})\right]\Big] d(t+\Delta t) \\
&+ \frac{1}{dt}\underbrace{\mathbb{E}\left[\left(\nabla_x V(\mathbf{x}_{t+\Delta t})\right)^T \sigma\left(\mathbf{x}_{t+\Delta t}, \alpha^*_{t+\Delta t}\right) dZ_{t+\Delta t}\right]}_{=0},
\end{aligned}$$

where the last term is zero due to the properties of Brownian motion. Taking the limit as $\Delta t$ goes to zero

$$\begin{aligned}
0 &= r(\mathbf{x}_t, \alpha^*_t) - \rho V(\mathbf{x}_t) + \left(\nabla_x V(\mathbf{x}_t)\right)^T f(\mathbf{x}_t, \alpha^*_t) \\
&+ \frac{1}{2}Tr\left[\left(\sigma(\mathbf{x}_t)\right)^T \Delta_x V(\mathbf{x}_t)\,\sigma(\mathbf{x})\right].
\end{aligned}$$

Finally, rearranging terms and makin use of the fatc that the policy is optimal, we obtain the HJB equation:

$$\rho V(\mathbf{x}) = \max_{\alpha} r(\mathbf{x}, \alpha) + \left(\nabla_x V(\mathbf{x})\right)^T f(\mathbf{x}, \alpha) + \frac{1}{2}Tr\left[\left(\sigma(\mathbf{x})\right)^T \Delta_x V(\mathbf{x})\,\sigma(\mathbf{x})\right].$$

## 7.2 Implementation Details

In this section, we briefly outline the various implementation choices that we made and alternative options available.

**Number of Neurons and Layers** There is no clear guidance in the deep learning literatur as to how many neurons or layers to use. The general recommendation is to start with a relatively low number of layers and increase them only if needed. We have experimented with different numbers of layers for all of our models and whilst the accuracy gains from two to three layers were substantial, a higher number of layers did not have similar effects. Thus, we employ a neural network with three hidden layers in all our models.

Until a few years ago, it was convention to reduce the number of neurons with each hidden layer. Recently, however, empirical evidence seems to suggest that a reduction of the number of neurons is only advantageous in deep convolutional networks that are mostly used for image recognition. For all other networks, it is now conventional to use the same number of neurons for each layer. We have experimented with both approaches and for our application we have achieved greater accuracy with the same number of neurons in each layer. In particular, for all neural networks in both versions of the neoclassical growth models we employ eight neurons per layer, whilst we employ 128 neurons in each layer in the dynamic capital allocation model.

**Activation Function** Considering the activation function, the most commonly used ones are the following:

- Logistic: $\phi(x) = \frac{1}{1+e^{-x}}$
- tanh: $\phi(x) = \tanh(x)$
- ReLU: $\phi(x) = \max\{0, x\}$
- leaky ReLU: $\phi(x) = \begin{cases} \alpha x, & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$

  $\alpha$ is either set to a conventional value of 0.01 or is also learned by the model in which case it is known as parametric leaky ReLU.
- ELU: $\phi(x) = \begin{cases} \alpha(e^x - 1), & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$.

Until a few years ago, sigmoidal functions like logistic or tanh were commonly used, since these are closest to the activation functions found in human brains. However, in many modern applications ReLU-style application functions are preferred. The reason for this is that the

gradients of sigmoidal activation functions zero out quickly, so that learning stops, while the gradients of ReLU-class functions do not die out. With sigmoidal activation functions one thus commonly observes "neuron death", meaning that where the gradients of a neuron are basically at zero, making further learning of this neuron all but impossible. Xu et al. (2015) evaluated different activation functions and generally find the following:

Logistic < tanh < ReLU < leaky ReLU < parametric leaky ReLU < ELU

For our applications on the other hand, tanh significantly outperforms all other activation functions. This is due to the smooth shape of tanh that corresponds better to the mostly smooth functions we are approximating. For this reason, we use tanh as our activation function in all of our applications.

**Weight Initialization** It is now widely accepted that the right initialization is fundamental for the convergence of the neural network. The reasoning behind this is quite simple. Since with sigmoidal activation functions gradients disappear for very large and small inputs, it is essential that the inputs are centered around zero. With traditional $N(0,1)$ initialization schemes, it has been shown empirically that we often observe vanishing (or sometimes exploding) gradients. The deeper in the network we go, the lower the gradients are so that no learning takes place there.

Glorot and Bengio (2010) propose an initialization scheme that counteracts this.[11] In particular, the authors propose that what is needed in order to avoid vanishing gradients is to have the variance of outputs equal the variance of the inputs for each layers and similarly to have the gradients to have equal variance before and after flowing through a layer. This ensures a smooth forward and backward flow through the network that is not diminishing the gradients ever further. Denoting by $n_{input}$ and $n_{output}$ the number of input and output connections of a layer, they argue that the optimal initialization is a normal distribution with mean 0 and standard deviation $\sigma = 4\sqrt{\frac{2}{n_{input}+n_{output}}}$. This is widely known as Xavier Initialization.[12]

**Input Normalization** To further reduce problems with vanishing gradients we employ input normalization, by standardizing each neural network input to have a mean of zero and a standard deviation of one. More advanced input normalization schemes like batch normalization did not lead to improvements for us and instead slowed down convergence significantly. This is due to the fact that with three layers our networks are not as exposed to the vanishing gradient problem as deeper networks and costly fixes are therefore not necessary for us.

[11] In fact, this paper was so successful that it is one of the key reasons for the renewed interest in neural networks.

[12] Note that this is only the optimal standard deviation for our tanh activation function. For other activation functions other formulas are available, such as He Initialization for ReLU activation functions.

**Optimizing Algorithm - Adam** Whilst we have used stochastic gradient descent for ease of exposition in our main algorithm, in our implementation we follow the literature in using the momentum-based Adam optimizer instead. The updating algorithm follows the five steps below:

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1)\nabla_\theta J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1-\beta_1^t}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1-\beta_2^t}$
5. $\theta \leftarrow \theta + \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \varepsilon}$

First, let us clear up some notation: $t$ denotes the iteration number (starting at 1). $\beta_1$ is the momentum decay hyperparameter, typically initialized to 0.9, whilst $\beta_2$ is the scaling decay hyperparameter, typically initialized to 0.999. $\varepsilon$ is usually initialized to a tiny number like $10^{-8}$. $\otimes$ denotes element-wise multiplication and $\oslash$ element-wise division. $J$ denotes the cost function, so $\nabla_\theta J(\theta)$ denotes the gradient of the cost function with respect to the weights $\theta$.

Next, let us quickly describe the basic principle of a momentum optimizer: Instead of relying solely on the current gradient like a gradient descent optimizer, it also takes into account previous gradients. It does this by subtracting the current gradient from a momentum vector $\mathbf{m}$ and then updating the weights by adding this momentum vector. In contrast to gradient descent, the gradient is thus not used as "speed" but instead as "acceleration".

Turning to Adam we can observe that this is the basic principle behind the algorithm. In step one, the momentum vector is updated by subtracting the gradient. Note that this is modified by using an average between the momentum and the gradient. As a result we care more about recent gradients than gradients further back. In the second step we update another vector $s$. This vector accumulates the square of the gradient of the cost function, i.e. a measure of the steepness of the cost function.[13] Again, we use an average to put more weight on recent gradients. Steps 3 and 4 are more of a technical detail. As $\mathbf{m}$ and $\mathbf{s}$ are initialized to zero, these steps help to boost $\mathbf{m}$ and $\mathbf{s}$ at the beginning of training. Step 5 is the crucial updating step. We update $\theta$ by adding the momentum times the learning rate $\nu$. However, the momentum vector is scaled down by the steepness of the accumulated gradients. This is included so that the optimizer does not just follow the steepest direction but instead takes into account all directions when updating. This significantly improves convergence by stopping the algorithm

[13]Note that methods using the full Hessian are not used in neural network learning as they are significantly slower.

from exploring "side valleys" and instead leading it towards its global minimum. $\varepsilon$ enters in order to avoid division by zero.

We have tried a variety of optimizers, but Adam yields by far the best results. Regarding the learning rate, $\eta = 0.001$ works well for us, even though the particular learning rate should not be as important here, as it is automatically updated by the algorithm itself over time. The conventional values of $\beta_1$ and $\beta_2$ work well for us. There are some examples in the literature where changing these parameters substantially improved the results, but our trials have not shown similar effects in our model.

**Backpropagation** One fundamental breakthrough in the deep learning literature is the backpropagation algorithm by Rumelhart et al. (1986). This algorithm allows an efficient training of the neural network as well as an efficient computation of all derivatives of the neural network with respect to its inputs. It is thus absolutely essential to our approach. The fundamental idea is that in addition to the computation of the neural network in a "forward pass", we can compute all partial derivatives in a "backward pass" and then combine those through the chain rule. For details, we refer the interest reader to the excellent exposition in Goodfellow et al. (2016).

## 7.3 Simulation Details for the Ergodic Distribution

In this section, we describe the exact implementation of the simulation of the ergodic distribution in our dynamic capital allocation model. The ergodic distribution is either given through closed-from expression or can be derived in an analogous way.

Start the procedure by specifying a time step $\Delta t$. Start an epoch $e$ with current neural network parameters $\Theta^e$. Now start the simulation at $I$ random starting points $\{\{z_{\ell,i,0}, k_{\ell,i,0}\}_{\ell=1}^{L}\}_{i=1}^{I}$. The simulation length is denoted by $T$. For each $i = 1 : I$, for each $t = 1 : T$, and for each $\ell = 1 : L$ recursively define:

$$z_{\ell,i,t} = z_{\ell,i,t-1} + (-\nu \log(z_{\ell,i,t-1}) + \frac{\sigma^2}{2}) z_{\ell,i,t-1} \Delta t + \sigma z_{\ell,i,t-1} \varepsilon_{\ell,i,t} \sqrt{\Delta t}, \qquad \text{where } \varepsilon_{\ell,i,t} \sim \mathcal{N}(0,1)$$

$$k_{\ell,i,t} = k_{\ell,i,t-1} + [\tilde{i}(\{k_{\ell,i,t-1}, z_{\ell,i,t-1}\}_{\ell=1}^{L}; \Theta^e) + \tilde{nk}(\{k_{\ell,i,t-1}, z_{\ell,i,t-1}\}_{\ell=1}^{L}; \Theta^e) - \delta k_{\ell,i,t-1}] \Delta t$$

Define the location and starting point specific simulation vectors:

$$z_{\ell,i} \equiv \begin{pmatrix} z_{\ell,i,1} \\ z_{\ell,i,2} \\ \vdots \\ z_{\ell,i,T} \end{pmatrix}, \qquad k_{\ell,i} \equiv \begin{pmatrix} k_{\ell,i,1} \\ k_{\ell,i,2} \\ \vdots \\ k_{\ell,i,T} \end{pmatrix}$$

Drop the first $B_1$ periods as burn-in in order to allow the simulation to move away from the initial random starting points into the ergodic set:

$$z_{\ell,i} \equiv \begin{pmatrix} z_{\ell,i,B_1+1} \\ z_{\ell,i,B_1+2} \\ \vdots \\ z_{\ell,i,T} \end{pmatrix}, \qquad k_{\ell,i} \equiv \begin{pmatrix} k_{\ell,i,B_1+1} \\ k_{\ell,i,B_1+2} \\ \vdots \\ k_{\ell,i,T} \end{pmatrix}$$

Combine the different starting points into two big data matrices:

$$z_\ell \equiv \begin{pmatrix} z_{\ell,1} \\ z_{\ell,2} \\ \vdots \\ z_{\ell,I} \end{pmatrix}, \qquad k_\ell \equiv \begin{pmatrix} k_{\ell,1} \\ k_{\ell,2} \\ \vdots \\ k_{\ell,I} \end{pmatrix}$$

$$Z \equiv \begin{pmatrix} z_1 & z_2 & \cdots & z_L \end{pmatrix}, \qquad K \equiv \begin{pmatrix} k_1 & k_2 & \cdots & k_L \end{pmatrix}$$

Split the data into $M$ mini-batches of size $N$, where $M = \frac{(T-B)\times I}{N}$:

$$\{Z^m, K^m\}_{m=1}^M \equiv \{Z[(m-1)\times N : m \times N, :], K[(m-1)\times N : m \times N, :]\}_{m=1}^M$$

For each $m \in \{1, ..., M\}$ define:

$$\{z_m^n, k_m^n\}_{n=1}^N = (Z^m[:], K^m[:])$$

Thus, we can update the weights for each $m \in \{1, ..., M\}$ (i.e. perform $M$ learning steps) based on $\{z_m^n, k_m^n\}_{n=1}^N$. After these $M$ learning steps, the neural network parameters will be updated from $\Theta^e$ to $\Theta^{e+1}$, with which we can enter the next simulation episode.

Regarding the choice of initial starting points in the next epoch we can take the last points from the previous epoch, since they are likely closer to the ergodic set than random starting points. That is $\{\{z_{\ell,i,0}^{e+1}, k_{\ell,i,0}^{e+1}\}_{\ell=1}^L\}_{i=1}^I = \{\{z_{\ell,i,T}^e, k_{\ell,i,T}^e\}_{\ell=1}^L\}_{i=1}^I$.

Finally, note that we only employ this simulation algorithm after a certain number of epochs. Before that we sample uniformly at random from the entire state space hypercube to get a good approximation for the policy functions. This is necessary since the policy functions are most likely very poor approximations in the beginning and simulation might thus actually move us away from the ergodic set instead of narrowing in on it.

## 7.4 Steady States

**Deterministic Steady State** The deterministic steady state is the steady state reached when eliminatin uncertainty from the model, that is $\sigma = 0$. We can then solve for the DSS in closed form: First note that without the stochastic component all locations are completely identical. Thus there is no incentive for the social planner to redistribute any capital between locations. The problem is therefore equivalent to solve 25 separate models. That is $nk_\ell = 0$ for all $\ell$ and we will focus on one location from now on. First note that with $\sigma = 0$, the steady state for the productivity process is determined by $\mu(z) = 0$, which leads to two steady states, $\bar{z} = 0$ and $\bar{z} = 1$. We will concentrate on the stable steady state $\bar{z} = 1$. To find the steady state for the capital stock, we consider the Hamiltonian of the problem:

$$H(k, i, \lambda) = u(zk^\alpha - i - \kappa_i i^2) + \lambda(i - \delta k)$$

The necessary and sufficient conditions for the solution are given by:

$$\begin{aligned} u'(c)(1 + 2\kappa_i i) &= \lambda \\ \dot{\lambda} &= \rho\lambda - u'(c)\alpha z k^{\alpha-1} + \lambda\delta \\ \dot{k} &= i - \delta k \end{aligned}$$

In steady state we then $\bar{z} = 1$ as well as $\dot{k} = \dot{\lambda} = 0$. Combining this with the equations above leads to the following nonlinear equation in $\bar{k}$:

$$\alpha \bar{z} \bar{k}^{\alpha-1} = (\rho + \delta)(1 + 2\kappa_i \delta \bar{k})$$

We can solve this equation with any nonlinear solution techninque such as bisection, which yields the DSS:

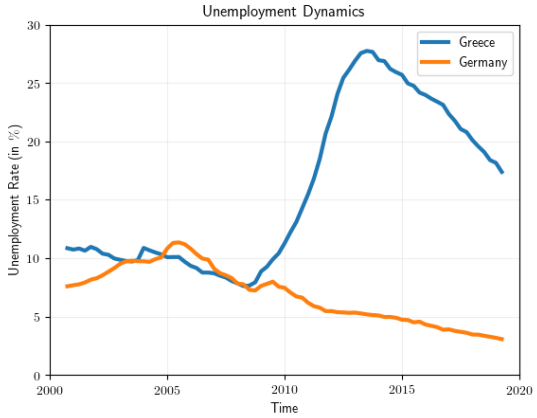
$$\bar{k}_{DSS} \approx 3.04$$

**Stochastic Steady State** The stochastic steady state is the steady state that is attained when uncertainty is present, i.e. $\sigma \neq 0$ and $\varepsilon_{\ell,t} = 0$ for all $\ell$ and all $t$. With $\varepsilon_{\ell,t} = 0$ and $\mu(z) = 0$, we can solve for the stochastic steady state for productivity in closed form: $z_{SSS} = e^{\frac{\sigma^2}{2\nu}} \approx 1.90$. In order to find the stochastic steady state for capital we utilize the policy functions for investment and net capital movement from our converged model and feed in a constant z at the level $z_{SSS}$ to the productivity of all locations. We find that capital in each location settles down to the steady state level $\bar{k}_{SSS} \approx 3.75$.

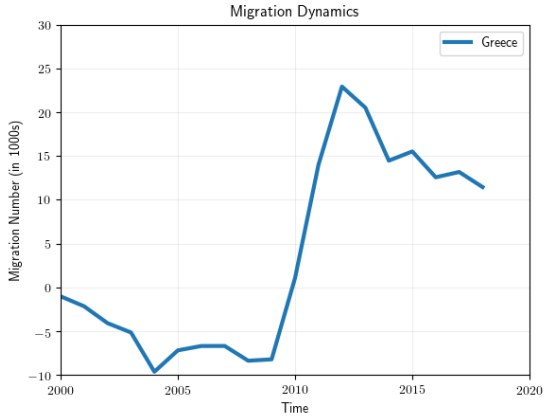
## 7.5 Empirical evidence on link between economic shocks and migration

One concrete setting where it is important to understand how economics shocks influence cross-country unemployment patterns is a freedom of movement zone such as the Schengen area. The literature on migration has highlighted that relative economic prospects are of first order importance in determining migration patterns. Below we show that this consideration is also of central importance in the context of the Schengen area during the Euro crisis of 2009.

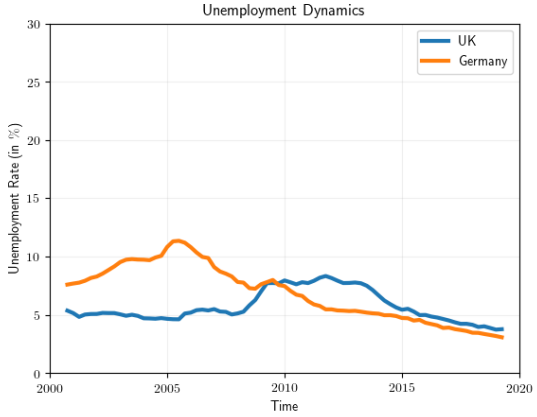
During this time countries that were the hardest hit by the crisis, such as Greece, saw a surge of outward migration to more prospering countries such as Germany. To illustrate this in more detail let us look at the case of Greece. Figure **??** shows the development of the unemployment rate in Greece and Germany. While unemployment in Greece surges, Germany remains relatively unaffected with only a minor increase in unemployment. Simultaneously panel B of Figure **??** demonstrates that while prior to the crisis there was a modest net flow of migrants from Germany to Greece, we see a substantial spike in the other direction during the crisis. In contrast to this, the UK only exhibits a modest uptick in unemployment, and therefore net migration from the UK to Germany only exhibits a modest increase. This indicates that the Greece-Germany pattern is indeed driven by the deteriorating conditions in Greece. Finally and most importantly, the pattern in Greece is economically significant, total net migration out of Greece over the five years following the start of the crisis amounted to about 20 to 30% of the peak unemployment rate at the time. This stresses the importance of including a realistic migration setting in a model that studies the propagation of shocks between countries.



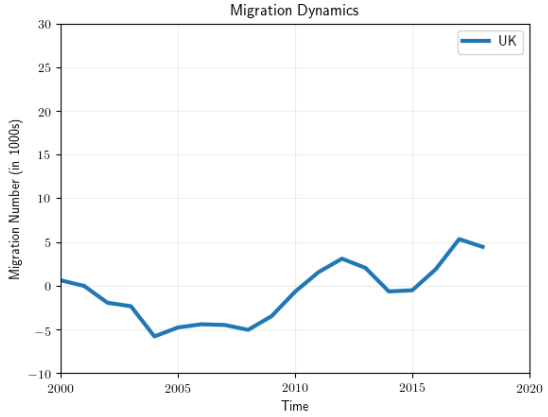

(a) Unemployment time series of Germany and Greece




(b) Net Migration from Greece to Germany




(c) Unemployment time series of Germany and the UK




(d) Net Migration from the UK to Germany

Figure 6: Unemployment and migration patterns during the European debt crisis. The left column shows the relative development of unemployment between Germany and Greece, see panel (a), as well between Germany and the UK, see panel (c). The right hand column shows the corresponding net migration flows. Specifically, panel (b) shows net migration from Greece to Germany, while panel (d) shows net migration from the UK to Germany.