Deep Learning for Solving Economic Models

Jesús Fernández-Villaverde*

The ongoing revolution in deep learning is reshaping research across many fields, including economics. Its effects are especially clear in solving dynamic economic models. These models often lack closed-form solutions, so economists have long relied on numerical methods such as value function iteration, perturbation, and projection techniques. Unfortunately, these approaches suffer from the curse of dimensionality, which makes global solutions computationally infeasible as the number of state variables increases. Deep learning offers a different approach: flexible tools that solve dynamic economic models by minimizing residuals in equilibrium conditions, and that can handle high-dimensional problems. This development promises to broaden the scope of quantitative economics. I illustrate the approach using the neoclassical growth model.
JEL: C61, C63, E27.

1. *Introduction*

The ongoing revolution in deep learning is transforming research across many disciplines, including economics. One area where its impact is especially pronounced is in solving dynamic economic models. Deep learning, a subset of artificial intelligence, has enabled the solution of equilibrium models that, until recently, were regarded as beyond tractability.

Equilibrium economic models (that is, specifications of preferences, technology, and information sets together with their equilibrium outcomes) often lack closed-form solutions. In other words, we cannot derive explicit formulae that map the model's primitives into agents' decisions, prices, and allocations. This difficulty is particularly severe when models are dynamic, stochastic, or incorporate agent heterogeneity.

* University of Pennsylvania, NBER, CEPR. This paper is based on a series of lectures delivered at Johns Hopkins University as a Distinguished Visiting Scholar. I am deeply grateful to the Department of Economics at Johns Hopkins for the invitation and invaluable feedback. Previous versions were presented at the Bank of Spain, Cambridge, ECB, Harvard, Northwestern, Oxford, Penn, Princeton, UC Santa Barbara, and Stanford. I thank all who offered comments on those occasions, the editor, David Romer, and the five referees. I also wish to thank my coauthors on the many papers I have written on machine learning in economics: Lorenzo Bretscher, Sara Casella, David Childers, Mahdi Ebrahimi Kahou, Kenneth Gillingham, Sebastián Gómez-Cardona, Stephen Hansen, Samuel Hurtado, Yilian Li, Joël Marbet, Galo Nuño, Jesse Perla, Omar Rachedi, Jan Rosa, Simon Scheidegger, Minchul Shin, Arnav Sood, George Sorg-Langhans, Maximilian Vogle, Le Xu, and Francesco Zanetti. Their collaboration has been instrumental in shaping my understanding of this topic. Naturally, all remaining errors are my own. The companion webpage `www.sas.upenn.edu/~jesusfv/deeplearning.html` will provide regular updates to the main text and code, as well as slide decks for teaching the material covered in this survey.

The standard response to the lack of closed-form solutions has been to use numerical approximation techniques. This area has expanded rapidly since the early 1970s, when the rational expectations revolution popularized dynamic equilibrium models, in which paper-and-pencil solutions are rare. Similar developments unfolded in industrial organization, with the rise of dynamic models of firm behavior and entry/exit; in finance, with the progress of modern equilibrium asset pricing; in international trade, with the spread of dynamic trade and spatial models; and in many other fields of economics.

To address this challenge, economists have employed methods such as value function iteration, perturbation (including linear and log-linearization as special cases), and projection methods (e.g., Chebyshev polynomials and finite elements), among others. A textbook treatment of these techniques is provided by Judd (1998).

Unfortunately, these techniques face a severe limitation. Characterizing the model's nonlinear dynamics, including stochastic volatility, occasionally binding constraints, non-stationarity, or equilibrium multiplicity, requires global solution methods such as value function iteration or projection techniques. But both approaches suffer from the curse of dimensionality (Bellman, 1957, p. IX), as the computational cost increases exponentially with the number of state variables, that is, the variables summarizing the past information needed to describe the system's dynamics.

For instance, suppose we implement value function iteration on a grid with n points per dimension. In one dimension, this requires n points; in two dimensions, n^2 points; in d dimensions, n^d points. With modest grid sizes (e.g., $n = 100$), models with more than five state variables quickly become computationally infeasible unless they possess a structure that is exploitable, such as suitability for sparse grids. In practice, however, the set of feasible global solutions remains sharply limited. Similar constraints apply to Chebyshev polynomials or finite elements.

The alternative is to use methods that can only yield locally accurate solutions, such as perturbations. While perturbations often provide sufficient accuracy in many problems of interest (for example, in solving standard New Keynesian models without a zero lower bound on the nominal interest rate), there are many cases where they do not. For instance, perturbations fail when evaluating business cycle models with rare disasters or, returning to the same example, when studying New Keynesian models in which the zero lower bound can bind stochastically.

The situation is frustrating: we end up working on the models we can solve, not the ones we would like to solve. Browsing through any recent volume of the top journals in economics reveals numerous passages in which authors apologize for the undesired simplifications they must impose to obtain numerical solutions.

Recent research, however, has demonstrated that deep learning can be applied to solving dynamic economic models. This class of methods can deliver accurate solutions over large regions of the state space, even in high-dimensional settings. Before introducing the main idea, let me first review some basic concepts.

Deep learning within artificial intelligence

Artificial intelligence is an expansive field encompassing a vast array of algorithms, from expert systems to generative adversarial networks and reinforcement learning (for a comprehensive textbook introduction, see Russell and Norvig, 2020). Figure 1 illustrates the argument using a Venn diagram. The largest set represents all artificial intelligence methods. Since the 1950s, computer scientists have developed approaches such as expert systems and symbolic artificial intelligence, which attempt to mimic expert knowledge and apply rules of inference and logic. While these methods are fascinating, their applications in solving dynamic economic models have been limited and will not be discussed further.[1]

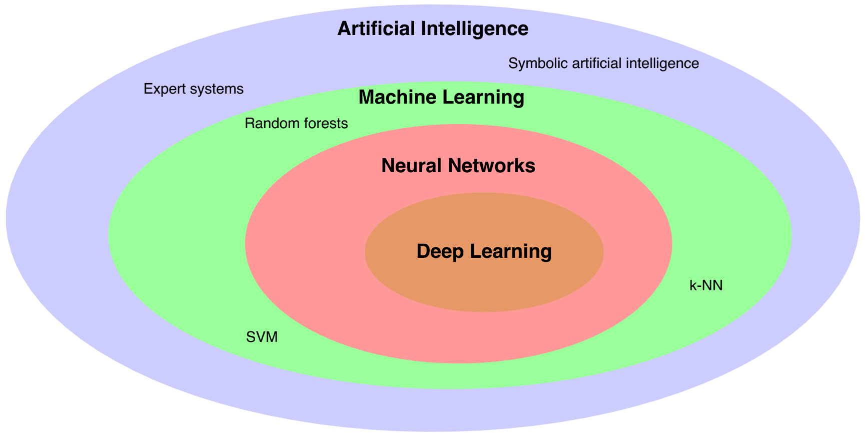

Figure 1. : Different methods in artificial intelligence

The first subset in Figure 1, machine learning, comprises algorithms that learn patterns from data. Examples include random forests, support vector machines (SVMs), and k-nearest neighbors (k-NN), although dozens of other machine learning methods exist.

The second subset comprises neural networks. This flexible parametric function approximation will be explored in detail in Section 4 as the building block for solving equilibrium models.

Finally, the third and smallest subset is deep learning, which involves the use of deep neural networks, that is, neural networks with many layers. Here "deep" just means many layers, as in "the well in my backyard is deep," not "deep" as in "*Sein und Zeit* is a deep book." Nearly all recent advances in solving equilibrium models I mentioned previously have been based on deep neural networks.

While the original work on neural networks dates back to the 1940s and 1950s (McCulloch and Pitts, 1943; Rosenblatt, 1958), the development of backpropagation by Rumelhart et al. (1986) enabled the practical training of multilayer networks. The current surge in interest began with Krizhevsky et al. (2012), who demonstrated the computational advantages of deep convolutional neural networks for large-scale image classification. Soon after, Mnih et al. (2015) demonstrated that deep reinforcement learning could achieve superhuman performance across a wide range of control tasks. Deep learning then spread rapidly across many fields. Nearly as importantly, Vaswani

[1]Interestingly, Herbert Simon, the 1978 Nobel Prize winner in Economics, was one of the three creators of the first artificial intelligence program, `Logic Theorist` (Newell et al., 1956). Perhaps, economists will one day revisit this tradition.

et al. (2017) introduced the transformer architecture, based on multi-head self-attention, which enabled the subsequent explosion of large language models.[2]

Deep learning and model solutions

When solving equilibrium models, we must approximate unknown functions. In a dynamic programming problem, for example, we want to find the value function $V(X)$, the return to an agent who follows the optimal decision rule given her states X, and the decision rule $d(X)$ itself.

In the language of deep learning, approximating an unknown target function at a collection of points is called *learning* the function. And because the approximation is carried out using a fixed algorithm, the process is referred to as *machine learning*.[3]

Traditional solution methods search for a parametric family $g_\theta(\cdot)$ to approximate the target:

$$V(X) \approx g_\theta(X).$$

In projection methods, the researcher specifies a *fixed basis* (for example, Chebyshev polynomials $\{T_m(X)\}$) and solves for the weights θ that best fit the target. The basis functions are chosen *ex ante*, before seeing the problem.

Neural networks take a different approach: instead of prescribing basis functions, they *learn* them. The approximation takes the form:

$$V(X) \approx g_\theta^{NN}(X) = v_0 + v^\top \phi(WX),$$

where W is a weight matrix to be learned, $\phi(\cdot)$ is an activation function applied element by element, v_0 is a scalar bias, v is a vector of output-layer weights, and $\theta = (v_0, v, W)$ collects all weights. The transformed features $z = \phi(WX)$, known as hidden units, form a *learned basis*.[4]

As with polynomial approximations, the final output is a linear combination of basis functions. The difference is that these bases are built during training by adjusting W, rather than being fixed in advance. A well-chosen basis makes the target function nearly linear in $\phi(WX)$: the basis absorbs the nonlinearities, leaving only a simple affine layer at the end. This enables us to handle problems in which the dimensionality of X is substantially larger than traditional methods can accommodate, and the approximations tend to generalize better to points outside the training set. There is a theory behind this claim. Barron (1993) shows that for target functions with suitable Fourier decay, single-hidden-layer networks achieve approximation rates provably sharper than any fixed-basis method using the same number of terms.

The idea of finding efficient representations is not new to economists. Consider the Cobb-Douglas production function $y = k^\alpha l^{1-\alpha}$. One of the first lessons we teach students is that, instead of working with levels, it is often easier to work with logs: $\log y = \alpha \log k + (1-\alpha) \log l$, since this representation is linear. Likewise, in growth models, we routinely rescale variables to remove trends and work with stationary

[2] Wooldridge (2021) provides a fast-paced history of artificial intelligence, warts and all.

[3] The algorithm is typically implemented on a computer for practical reasons, but the terminology refers to the automation of the procedure rather than to the machine itself.

[4] To avoid excessive notation at this stage, I present the argument using a single-hidden-layer network. I extend it to deep networks with multiple hidden layers later.

representations. Deep learning searches for analogous transformations when intuition fails. What is the efficient representation of states in a heterogeneous New Keynesian economy, or in an integrated assessment climate model with inherently non-stationary dynamics? Thanks to multiple layers of hidden units, deep learning automatically constructs these representations. These learned bases are also sometimes interesting in their own right, whereas a single Chebyshev coefficient rarely means much by itself.

Learned bases are not a substitute for analytic solutions when these exist; analytic solutions are almost always better for their transparency and sharp comparative statics. The question is: when no analytic solution is available or when it is too complex to use, how can we build a numerical solution that is both feasible and efficient?

Finally, as I will discuss how applying deep learning to solve dynamic equilibrium models is not a straightforward import from computer science: the evaluation points at which we train the network are themselves generated by the equilibrium we are trying to compute, a circularity I call the *equilibrium loop*.

A vibrant ecosystem

The numerous strengths of deep learning have fostered a vibrant ecosystem of hardware and software. Economic history reminds us that methods often succeed less because of intrinsic technological superiority than because of the ecosystems that support them (David, 1985).

On the hardware side, deep learning algorithms are embarrassingly parallelizable, fueling the rise of manufacturers such as NVIDIA, whose graphical processing units are optimized for this type of parallelism.

On the software side, researchers benefit from powerful open-source libraries such as `PyTorch` and `JAX`, which abstract away backend complexities and allow economists to concentrate on the substance of their problems. Consequently, deep learning code is often simple, readable, and easy to maintain.

This thriving ecosystem has attracted substantial human capital and investment from both academia and industry. As a result, rapid advances in hardware, software, and algorithms are likely to occur in the near future, providing a vivid illustration of directed technological change.

A few references

A thorough treatment of how deep learning can be used to solve equilibrium models in economics would require a full monograph, if such a monograph can even be written in a field evolving this rapidly. Given the space limitations, the reader should view the following pages as a brief introduction to deep learning as a solution method.

For those who want to go beyond these resources, Murphy (2022, 2024) offers an encyclopedic treatment of machine learning that is appealing for economists because of the books' emphasis on a probabilistic interpretation.[5] Focusing more narrowly on deep neural networks, Prince (2023) provides outstanding insights, while Aggarwal (2023) is comprehensive and up to date. Zhang et al. (2023) is valuable for the practical computational aspects of deep neural networks. Sutton and Barto (2018)

[5]Furthermore, and very much in the spirit of the field, both volumes are open source and available at `https://probml.github.io/pml-book/`.

remains the best introduction to reinforcement learning, a related solution method that, unfortunately, I must omit here.

Other surveys of machine learning in economics, going beyond the application of deep learning to solving models, include Chakraborty and Joseph (2017), Athey and Imbens (2019), Scheidegger and Bilionis (2019), Nagel (2021), Kelly and Xiu (2023), de Araujo et al. (2024), and Dell (2025).

A roadmap

The paper proceeds as follows. Section 2 sets up the general problem of solving equilibrium models via function approximation. Section 3 covers approximation methods with fixed bases and discusses their limitations. Section 4 introduces single-hidden-layer neural networks, which provide the foundation for the rest of the paper. Section 5 extends this framework to deep neural networks with multiple hidden layers. Section 6 describes how neural networks are trained, while Section 7 outlines the basic principles of architecture design. Sections 8 and 9 illustrate how these ideas apply to more general dynamic economic models. Section 10 concludes.

2. *Solving equilibrium economic models via function approximation*

When an analytic solution is unavailable (or too complex for practical use), solving an equilibrium economic model reduces to approximating unknown functions that map the model's state variables into equilibrium objects, such as allocations and prices. A non-exhaustive list of such cases includes:

1) In macroeconomics, the value function of agents' dynamic programming problems and the associated policy functions mapping current states into controls and next-period states.

2) In industrial organization, firms' continuation values as functions of their states, which determine entry and exit decisions as well as pricing, investment, R&D, and product development.

3) In labor economics, finite-horizon dynamic problems describing life-cycle decisions about education, labor supply, retirement, savings, household formation, fertility, and health behavior.

4) In search and matching models, objects such as vacancy posting rules, wage-setting functions, and market tightness.

5) In game theory, the best response functions of players.[6]

6) In international trade, functions allocating comparative advantage across firms and countries.

7) In public finance, optimal tax functions (labor income, capital income, or consumption taxes) as functions of states such as productivity, wealth, or age, often in heterogeneous-agent models.

[6]Most of the ideas in this paper extend to the approximation of correspondences, such as best-response correspondences or the correspondence of Nash or sequential equilibria. For simplicity, I abstract from this generalization.

8) In political economy, voting rules or political support functions mapping economic states into equilibrium policy outcomes.

9) In development economics, occupational choice and technology adoption rules mapping household or firm states into sectoral allocation, investment, and migration decisions.

10) In urban and regional economics, location choice, land-use, and commuting functions mapping household or firm characteristics into spatial equilibria.

11) In asset pricing, the pricing kernel of the economy as a function of the state.

12) In corporate finance, optimal asset and liability choices that maximize discounted firm value.

1. A common mathematical structure

All the problems enumerated above share a common mathematical structure. The goal is to approximate an unknown target function

$$y = f(X),$$

where $y \in \mathbb{R}$ is a scalar output and $X = (1, x_1, x_2, \ldots, x_N)$ is a vector of input features (e.g., the state variables of the model).

For most of the discussion, the non-constant components $(x_1, \ldots, x_N)$ are taken to belong to a feature space $\mathcal{X} \subseteq \mathbb{R}^N$, although $\mathcal{X}$ may be more general. For example, it may include discrete variables, bounded domains, or other constraints. Nothing essential in the analysis hinges on this distinction.

The dimension N may be large, possibly in the tens of thousands. This situation arises naturally in dynamic programming problems with many state variables, such as life-cycle models, corporate finance applications, or in models with heterogeneous agents.

I assume that the output y is scalar to keep notation light; vector-valued objects (policy vectors, price vectors, probability distributions representing mixed strategies in game theory) are handled by stacking outputs.

I include a constant among the components of X for notational convenience. This simplifies the treatment of intercepts in linear combinations and affine approximations used below. If the underlying function does not require an intercept, the constant can be interpreted as a dummy input feature with zero weight.

In applications to dynamic economic models, $f(\cdot)$ is typically deterministic. There are, however, settings in which one approximates functions observed with noise,

$$y = f(X) + \epsilon,$$

where ϵ is a random variable satisfying $\mathbb{E}[\epsilon \mid X] = 0$. One example arises in stochastic games with mixed strategies, where payoffs depend on the realization of opponents' randomized actions. In that case, ϵ reflects strategic uncertainty, even though the expected payoff function is deterministic.

2. *The evaluation points*

When approximating the target function $f(\cdot)$, we work with a finite set $\{X_l\}_{l=1}^{L}$ of input features, which I refer to as the evaluation points. In the context of solving dynamic economic models, $\{X_l\}_{l=1}^{L}$ typically consists of grid points for the model's state variables or simulated endogenous variables (for the X_l), rather than observations from the real world, as would be the case in econometrics.[7]

This setup has clear advantages and disadvantages. On the one hand, L is typically a choice variable for the researcher (e.g., the number of grid points, the length of the simulation), thereby removing concerns about too few evaluation points, a common problem in many econometric applications where researchers aim to approximate an empirical function with limited real-world data. A researcher should choose the value of L to balance the accuracy of the solution (a higher L typically yields greater accuracy) and the computational time (a higher L typically implies more computation time, everything else equal).

On the other hand, the researcher must decide where to place the grid points or how to generate simulated paths for endogenous variables. Poor choices along these dimensions lead to poor solutions. Note that this issue is not specific to deep learning. It also arises in projection methods or value function iteration, where one must choose collocation or grid points. In fact, neural networks are often more resilient to poor choices of these points than other numerical methods.

3. *A basic example: The deterministic neoclassical growth model*

To illustrate how one can approximate an unknown function at some evaluation points, let me consider one of the simplest examples of a dynamic economic model that needs to be solved numerically: the deterministic neoclassical growth model.[8]

A social planner picks a sequence of consumption, $\{c_t\}_{t=0}^{\infty}$, to solve the discounted utility of the infinitely lived representative household:

$$\max_{\{c_t\}} \sum_{t=0}^{\infty} \beta^t \log(c_t),$$

subject to the resource constraint:

$$c_t + k_{t+1} = k_t^{\alpha} + (1-\delta)\, k_t, \text{ given } k_0. \tag{1}$$

Here, $\beta < 1$ is the discount factor, k_t is capital, $0 < \alpha < 1$ is the curvature of the production function with respect to capital (labor is constant and normalized to one), and $0 \leq \delta < 1$ is the depreciation rate. The state variables of the social planner's problem are $X_t = (1, k_t)$. As before, the constant 1 appears as a state variable for notational convenience.[9]

[7] I use the term *evaluation points* rather than *training set* because the same framework applies to projection methods, where "training set" is not standard terminology. Later, when discussing neural networks, I split the evaluation points into a training set and a held-out test (or validation) set.

[8] See, for a detailed introduction to this model, Stokey et al. (1989) and slide deck 2 in `www.sas.upenn.edu/~jesusfv/deeplearning.html`.

[9] At the cost of some extra notation, all the examples would go largely unchanged if I were dealing with the competitive equilibrium.

An optimal solution to this sequential problem must satisfy the Euler equation for consumption

$$\frac{1}{c_t} = \beta \frac{1}{c_{t+1}} \left(\alpha k_{t+1}^{\alpha-1} + 1 - \delta \right)$$

and the transversality condition $\lim_{T\to\infty} \beta^T c_T^{-1} k_{T+1} = 0$.

Associated with the sequential social planner's problem, there is an equivalent recursive formulation of the problem with a Bellman equation on k:

$$\begin{aligned} V(k) &= \max_c \{\log(c) + \beta V(k')\} \\ \text{s.t. } k' &= k^\alpha + (1-\delta)\, k - c, \end{aligned} \tag{2}$$

where $V(\cdot)$ is the value function. I use recursive notation, omitting time subscripts and denoting next-period variables with a prime.

The input features in this example are $\{X_l\}_{l=1}^L = \{(1, k_l)\}_{l=1}^L$, where $\{k_l\}_{l=1}^L$ are grid points over k. For instance, we can define a uniform grid from $k = 0.001$ to an upper bound $k = k_{\max}$ that the social planner would never reach given k_0.[10] Below, I will show a case where $\{X_l\}_{l=1}^L$ comes from a simulation.

Then, let us define a candidate solution as a parametric function $\widehat{V}_\theta(k)$ belonging to some approximating family (e.g., polynomials or neural networks), where θ denotes the weights to be determined. The Bellman equation residual (or error) for this candidate approximation is:

$$\mathcal{R}[\widehat{V}_\theta](k) = \widehat{V}_\theta(k) - \max_{k'} \left\{ \log\left(k^\alpha + (1-\delta)k - k'\right) + \beta \widehat{V}_\theta(k') \right\}, \tag{3}$$

where I have used the resource constraint (1) to eliminate c.

Numerical approximation procedures aim to find weights θ that minimize a measure of the residuals $\mathcal{R}[\widehat{V}_\theta](k_l)$ across $\{k_l\}_{l=1}^L$. I will return to discussing appropriate metrics in Subsection 2.5, but for now, the mean of squared residuals is a good default choice. Under appropriate technical conditions, the minimizer provides a global approximation to the true value function over $[0.001, k_{\max}]$.

4. Extending the basic example

The deterministic neoclassical growth model illustrates the core idea of solving equilibrium models through function approximation, but most applications involve stochastic environments and different types of equilibrium conditions. I now extend the example by introducing productivity shocks.

The resulting stochastic neoclassical growth model has a new social planner's problem:

$$\max_{c_t} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u\left(c_t\right),$$

[10]The steady-state capital of this model is $k_{ss} = \left(\frac{1}{\alpha}\left(\frac{1}{\beta} - 1 + \delta\right)\right)^{\frac{1}{\alpha-1}}$. Since for any $k_0 > k_{ss}$, we can prove analytically that $k_0 > k_1$, we only need to make $k_{\max} > k_0$.

subject to:

$$c_t + k_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta)\, k_t$$
$$z_t = \rho z_{t-1} + \sigma \varepsilon_t, \quad \varepsilon_t \sim \mathcal{N}(0,1),$$

given k_0 and z_0.

In addition to the previously introduced notation, $\mathbb{E}_0$ is the expectation conditional on time-0 information, z_t denotes the log of productivity, ε_t denotes the productivity shock, and, to illustrate generality, I have switched to a constant relative risk aversion (CRRA) utility function

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma},$$

where $\gamma > 0$ is the coefficient of relative risk aversion, with the convention that $u(c) = \log(c)$ when $\gamma = 1$. The state variables are $X_t = (1, k_t, z_t)$ and the transversality condition the optimal solution must satisfy is $\lim_{T\to\infty} \mathbb{E}_0 \left[\beta^T u'(c_T) k_{T+1}\right] = 0$.

With this version of the model as a running example, I now present three approaches to constructing residuals for evaluating candidate solutions. Formally, these three approaches are examples (but not an exhaustive list) of a large class of problems in economics that search for a function d that solves a functional equation $\mathcal{T}(d) = \mathbf{0}$, where $\mathcal{T}$ is an operator mapping between function spaces and $\mathbf{0}$ denotes the zero function.

Each approach yields a distinct functional equation, and the residual quantifies the extent to which the candidate fails to satisfy the equilibrium conditions. The choice of formulation depends on the specific problem; all three are adaptable to a wide range of dynamic equilibrium models. In each case, the transversality condition must also be checked.

Value functions

This approach generalizes the deterministic case of Subsection 2.3. We work with the recursive formulation of the problem:

$$V(X_t) = \max_{k_{t+1}} \left\{ u(c_t) + \beta \mathbb{E}_t V(X_{t+1}) \right\},$$

where $c_t = e^{z_t} k_t^{\alpha} + (1-\delta) k_t - k_{t+1}$.

To find $V(X_t)$, I can propose a candidate solution $\widehat{V}_\theta(X_t)$, which, as before, belongs to some parametric class, and define the Bellman equation residual:

$$\mathcal{R}[\widehat{V}_\theta](X_t) = \widehat{V}_\theta(X_t) - \max_{k_{t+1}} \left\{ u(c_t) + \beta \mathbb{E}_t \widehat{V}_\theta(X_{t+1}) \right\},$$

where $c_t = e^{z_t} k_t^{\alpha} + (1-\delta) k_t - k_{t+1}$.

As in the deterministic case, we search for θ that minimizes a measure of the residuals across evaluation points (see the discussion about the *equilibrum loop* below).

Decision rules

The second approach is to work with the Euler equation for the social planner:

$$u'\left(c_t\right) - \beta \mathbb{E}_t\left\{u'\left(c_{t+1}\right)\left(1 + \alpha e^{z_{t+1}} k_{t+1}^{\alpha-1} - \delta\right)\right\} = 0. \tag{4}$$

This approach is useful when working with the Euler equation is more convenient than working with the recursive formulation.[11]

A solution in this approach is a decision rule (also known as a policy function) $c_t = d(X_t)$ that satisfies the Euler equation:

$$u'(d(X_t)) = \beta \mathbb{E}_t\left\{u'(d(X_{t+1}))\left(1 + \alpha e^{z_{t+1}} k_{t+1}^{\alpha-1} - \delta\right)\right\}.$$

and where $k_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta)\, k_t - d(X_t)$ comes from the resource constraint of the economy.

To find a candidate solution $\widehat{d}_\theta(X_t)$, we define an Euler equation residual:

$$\mathcal{R}[\widehat{d}_\theta](X_t) = u'(\widehat{d}_\theta(X_t)) - \beta \mathbb{E}_t\left\{u'(\widehat{d}_\theta(X_{t+1}))\left(1 + \alpha e^{z_{t+1}} \widehat{k}_{t+1}^{\alpha-1} - \delta\right)\right\},$$

where $\widehat{k}_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta)\, k_t - \widehat{d}_\theta(X_t)$ approximates next-period capital. As before, we search for θ that minimizes a measure of the residuals $\mathcal{R}[\widehat{d}_\theta](X_t)$.

Conditional expectations

Alternatively, we can directly approximate the conditional expectation that appears in the Euler equation, an idea known as the *parameterized expectations algorithm* (Den Haan and Marcet, 1990). This approach is useful when the decision rule has kinks or discontinuities. A classic example is the zero lower bound on nominal interest rates in a New Keynesian model: the kink in the central bank's decision rule makes it difficult to approximate, whereas the conditional expectation of marginal utility remains smooth.

If I define $d(X_t) = \mathbb{E}_t\{u'(c_{t+1})(1 + \alpha e^{z_{t+1}} k_{t+1}^{\alpha-1} - \delta)\}$, I can find

$$c_t = (u')^{-1}(\beta d(X_t))$$

where $(u')^{-1}(\cdot)$ denotes the inverse marginal utility function and then use the resource constraint to get $k_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta)\, k_t - c_t$.

To find $d(X_t)$, I can propose a candidate solution $\widehat{d}_\theta(X_t)$ and simulate the economy forward for L periods: at each step, I compute c_t from the Euler equation, k_{t+1} from the resource constraint, and draw $\varepsilon_{t+1} \sim \mathcal{N}(0,1)$ to obtain $z_{t+1} = \rho z_t + \sigma \varepsilon_{t+1}$ and form $X_{t+1} = (1, k_{t+1}, z_{t+1})$.

I then define the conditional expectation residual:

$$\mathcal{R}[\widehat{d}_\theta](X_l) = \widehat{d}_\theta(X_l) - u'(c_{l+1})\left(1 + \alpha e^{z_{l+1}} k_{l+1}^{\alpha-1} - \delta\right),$$

[11]For instance, if instead of working with the social planner, we are solving for a market equilibrium that is not Pareto efficient, using the Euler equations is usually much easier.

and search for θ that minimizes the mean square of these residuals over the L periods of the simulated path. With the updated θ, I re-simulate the economy, recompute the residuals, and iterate until convergence. The key observation is that the realized value $u'(c_{l+1})(1+\alpha e^{z_{l+1}}k_{l+1}^{\alpha-1}-\delta)$ is a draw from the distribution whose expectation is $d(X_l)$, so minimizing the squared residuals amounts to a regression that consistently estimates the conditional expectation without numerical integration.

5. *The equilibrium loop*

The previous discussion of how we minimize the conditional-expectation residual suggests a more general point. Given any residual $\mathcal{R}[\widehat{f}_\theta](X)$ from an economic model and an approximating function $\widehat{f}_\theta(X)$, the natural goal would be to solve:

$$\theta^* = \arg\min_\theta \mathbb{E}_{\xi^*(\cdot)}\mathcal{R}[\widehat{f}_\theta](X),$$

where the expectation is taken with respect to the ergodic distribution $\xi^*(\cdot)$ of the states X (for deterministic models, one can instead minimize over transitional paths from given initial conditions). By weighting the residual with $\xi^*(\cdot)$, we concentrate accuracy where the economy actually spends its time.

If we knew $\xi^*(\cdot)$, the solution would be straightforward. We would just sample L points $\{X_l\}_{l=1}^L$ from $\xi^*(\cdot)$ and solve:

$$\theta^* = \arg\min_\theta \frac{1}{L}\sum_{l=1}^{L}\mathcal{R}[\widehat{f}_\theta](X_l).$$

The difficulty is that $\xi^*(\cdot)$ is an endogenous object that we do not know before we solve the model. This is the typical situation in economics. This *equilibrium loop* (i.e., the correct evaluation points are generated by the model we want to solve) is one of the main reasons why solving dynamic equilibrium models requires non-trivial changes to algorithms in computer science or engineering. This is a subtle yet crucial point that is often overlooked by researchers (and, in my experience, a point that many non-economists struggle with).

If the model is sufficiently low-dimensional, we can construct a grid over a wide range of the state space and minimize the loss function on this grid (either using a sup norm or Euclidean distance or by requiring that the residuals vanish exactly at selected grid points, an approach known as collocation), thereby avoiding the problem (although at the cost of some computational efficiency). But this approach becomes infeasible in high dimensions. A grid with G points per dimension requires G^N points in total, and most of them will lie in economically irrelevant regions of the state space.

A simple solution is to iterate between sampling and optimization, periodically regenerating the evaluation points as $\widehat{f}_\theta(X)$ is refined. Each round of regeneration and re-optimization is what I call an *epoch*:

1) Guess an initial θ^0 (the superscript will indicate the epoch index).

2) Generate an epoch $\{X_l^0\}_{l=1}^L$ by simulating from the model using the decision rule implied by θ^0. If we are approximating a decision rule, we use the candidate

decision rule evaluated at θ^0. If we are approximating another object (e.g., a conditional expectation), we find the decision rule associated with the candidate conditional expectation evaluated at θ^0.

3) Re-solve the model using this epoch to obtain θ^1 and, with this θ^1, generate a new epoch $\{X_l^1\}_{l=1}^L$.[12]

4) Iterate until convergence, i.e., until a metric of the residual (i.e., the sup norm or Euclidean distance over the evaluation points) is sufficiently small.

Although this algorithm usually converges, there is no theoretical guarantee that it will. Therefore, a good initial guess and a carefully designed sampling of evaluation points are essential. We will return to this point below.

6. Selecting the candidate solution class

Given any of the three residual formulations above and the equilibrium loop, the remaining choice is the approximation family used for the candidate solution. I will consider two approaches: fixed bases (polynomials) and learned bases (neural networks).

3. Function approximation with fixed bases

The classical approach to constructing a candidate solution for the three approaches above is to use a fixed basis of polynomial functions (including monomials). As I argue, these methods perform well on low-dimensional problems but scale poorly as the number of state variables increases, a limitation that neural networks are designed to overcome.

1. A linear approximation

A simple candidate for approximating a target function $f(X)$ given some evaluation points is a linear combination of X:

$$f(X) \approx g_\theta^{LA}(X) = \theta_0 + \sum_{n=1}^{N} \theta_n x_n, \tag{5}$$

where $\theta = (\theta_0, \ldots, \theta_N)$ are weights chosen to make the approximation close to the target output under an appropriate metric.

For instance, in the example of the deterministic neoclassical growth model above, one may define the candidate solution

$$\widehat{V}(k) = \theta_0 + \theta_1 k,$$

plug it at the Bellman residual equation (3) at the set of grid points $\{(1, k_l)\}_{l=1}^L$, and select θ to minimize the mean squared Bellman residual, $\frac{1}{L}\sum_{l=1}^{L} \mathcal{R}[\widehat{V}_\theta](k_l)^2$.

[12]This regeneration might include adaptive schemes, e.g., sampling more where residuals are currently high. This may be important for accurately solving the model in the tails of the ergodic distribution or in counterfactual regions where nonlinearities are pronounced.

While linear approximations are often useful, they fail to capture the nonlinear relationships central to many models. For example, we know that, since we have assumed a concave period utility function ($\log(\cdot)$) in the deterministic neoclassical growth model, the true value function $V(\cdot)$ is also concave on k.

This inherent nonlinearity in the functions we seek to approximate is common. Decision rules and conditional expectations in dynamic programming problems, best responses in game theory, vacancy posting and wage-setting rules in search and matching, and pricing kernels in finance are rarely linear. And choice probabilities in discrete choice models are inherently nonlinear functions of the state variables.

2. Incorporating nonlinear terms

A natural way to introduce nonlinearities into the approximation (5) is to add monomials of X. For instance, including quadratic terms on X yields a quadratic approximation:

$$f(X) \approx g_\theta^{QA}(X) = \theta_0 + \sum_{n=1}^{N} \theta_n^1 x_n + \sum_{n=1}^{N} \theta_n^2 x_n^2 + \sum_{n=1}^{N-1} \sum_{m=n+1}^{N} \theta_{n,m} x_n x_m. \tag{6}$$

In the Bellman equation (2) above, the quadratic candidate solution becomes:

$$\widehat{V}(k) = \theta_0 + \theta_1 k + \theta_2 k^2.$$

As before, we can substitute this quadratic approximation for the Bellman residual equation (3) at the set of grid points $\{k_l\}_{l=1}^{L}$, and select θ to minimize the mean squared Bellman residual, $\frac{1}{L}\sum_{l=1}^{L} \mathcal{R}[\widehat{V}_\theta](k_l)^2$.

Despite its intuitive appeal, adding monomial terms quickly runs into limitations. Low-order terms such as x_n^2 differ substantially from x_n and expand the *capacity* (or expressive power) of the approximation. However, on bounded domains, high-order monomials such as x_n^{10} and x_n^{11} become nearly collinear, contributing little additional flexibility while severely degrading numerical conditioning. In particular, the weights θ_n^{10} and θ_n^{11} cannot be determined stably: small changes in $\{X_l\}_{l=1}^{L}$ or in the floating-point arithmetic (e.g., the order of operations in the code) can generate large changes in the approximated function $g_\theta^{QA}(X)$.

Chebyshev polynomials of the first kind offer a remedy. Because they are orthogonal on $[-1, 1]$ with respect to the weight $(1 - x^2)^{-1/2}$, even high-order Chebyshev terms remain nearly orthogonal when evaluated on a grid, dramatically improving numerical conditioning relative to monomials.

The multivariate approximation with Chebyshev polynomials takes the form:

$$f(X) \approx g_\theta^{CP}(X) = \theta_0 + \sum_{m \in \mathcal{M}} \theta_m \prod_{j=1}^{N} T_{m_j}(x_j), \tag{7}$$

where $T_{m_j}(\cdot)$ is the univariate Chebyshev polynomial of order m_j, $m = (m_1, \ldots, m_N)$ is a multi-index, and $\mathcal{M} \subset \mathbb{N}_0^N$ specifies which terms to include. Common choices are the total-degree set $\mathcal{M} = \{m \in \mathbb{N}_0^N : |m|_1 \leq D\}$, which limits the sum of exponents, and the tensor-product set $\mathcal{M} = \{0, \dots, D\}^N$, which includes all combinations up to order D in each dimension.[13]

Returning to our Bellman equation example, we can define the candidate solution:

$$\widehat{V}_\theta(k) = \theta_0 + \sum_{m \in \mathcal{M}} \theta_m T_m(k),$$

plug it into equation (3) at a set of collocation points $\{k_l\}_{l=1}^{L}$ (e.g., the roots of the $(D+1)$-th Chebyshev polynomial defined over $[k_0, k_{max}]$).[14]

A standard practice is to select θ to make the Bellman residual exactly zero at those collocation points: $\mathcal{R}[\widehat{V}_\theta](k_l) = 0$. That is, collocation projects the solution onto a Chebyshev polynomial basis by enforcing the Bellman equation exactly at selected nodes. Under standard regularity conditions, the rest of the approximation will converge as well to the true unknown function over $[k_0, k_{max}]$ as the number of collocation points increases.

3. *The curse of dimensionality*

Finding θ in equation (7) via collocation projection suffers severely from the curse of dimensionality: once X has five or more dimensions, computational costs become prohibitive, even with greedy techniques such as sparse grids (Brumm and Scheidegger, 2017). Models with heterogeneous agents, where the number of state variables grows rapidly, are particularly challenging.

Economists have responded to this challenge either by solving models with strong simplifying assumptions or by solving them over a small subset of the state space. For example, we can implement a higher-order perturbation around the model's deterministic steady state, or solve the model in sequence space to obtain a concrete realization of the exogenous shocks.[15]

However, both alternatives are unsatisfactory in many applications in which we seek to dispense with simplifying assumptions (e.g., when providing quantitative policy recommendations) or aim to characterize solutions across a wider range of the state space.

How can we tackle the curse of dimensionality and solve larger, richer equilibrium models? Neural networks offer a promising alternative by learning representations of X rather than relying on prescribed bases. As I discuss in Section 4, there are, in fact, two distinct aspects of the curse (one concerning approximation, the other the availability of evaluation points), and neural networks address each differently.

4. *A single-hidden-layer neural network*

A *single-hidden-layer* neural network is a parametric class of functions with exactly one layer of nonlinear transformations followed by a linear output layer.[16]

[13] See Fernández-Villaverde et al. (2016) for an introduction to the use of Chebyshev polynomials in solving economic models.

[14] Collocation points are a particular choice of evaluation points used in projection methods.

[15] See, again, Fernández-Villaverde et al. (2016) for an introduction to higher-order perturbation and Auclert et al. (2021) for sequence space methods.

[16] Single-hidden-layer networks are sometimes called shallow neural networks. The term *shallow*, however, is used inconsistently in the literature and may also refer to networks with a small number of hidden layers (two or three), rather than exactly one.

Rather than directly constructing an approximation that is nonlinear in the input features,

$$f(X) \approx g_\theta(X),$$

as in equations (6) and (7), a single-hidden-layer network first transforms the inputs through a hidden layer and then combines the results linearly.

More specifically, the input features X are first mapped into transformed features $\phi(WX)$, and the approximation is then linear on these transformed features:

$$f(X) \approx g_\theta^{NN}(X) = v_0 + v^\top \phi(WX),$$

where $W \in \mathbb{R}^{M\times(N+1)}$ is the matrix of hidden-layer weights, $\phi : \mathbb{R} \to \mathbb{R}$ is a nonlinear activation function applied element by element, v_0 is the output bias, and $v \in \mathbb{R}^M$ is the vector of output-layer weights. Finally, the tuple $\theta = (v_0, v, W)$ collects all network weights.

As Section 6 will explain in detail, the weight matrix W is *trained* so that the features $\phi(WX)$ form a basis adapted to the problem at hand. Unlike the fixed-basis methods of Section 3, neural networks offer a constructive procedure to learn representations that minimize approximation error in the candidate solution.

Returning to our example in Section 2, we search for an approximation of the value function of the form:

$$\widehat{V}(k) = v_0 + v^\top \phi(W(1, k)).$$

Although compact, this expression encodes a three-step construction. Making these steps explicit helps us understand both how neural networks work and why they are effective.

Step 1: Preactivations.

Recall that $X \in \mathbb{R}^{N+1}$ denotes the vector of input features, where the first component is a constant equal to one.

Define the weight matrix of the hidden layer as

$$W \equiv \begin{pmatrix} w_{1,0} & w_{1,1} & \cdots & w_{1,N} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ w_{M,0} & w_{M,1} & \cdots & w_{M,N} \end{pmatrix} \in \mathbb{R}^{M\times(N+1)}.$$

With this weight matrix, we map the input features into a vector of M *preactivations*

$$a = WX,$$

where $a \in \mathbb{R}^M$. The "+1" in the dimension of W accounts for the constant term included in X, which absorbs the bias into the weight matrix.

These transformations are simply matrix multiplications and are computationally inexpensive, scaling well even when the number of input features is large. Moreover, they are trivially parallelizable, making them particularly well-suited to modern hardware architectures.

This structure is easy to see by writing each component a_m, for $m = 1, \ldots, M$, explicitly as an affine transformation of the input features:

$$a_m = w_{m,0} + \sum_{n=1}^{N} w_{m,n} x_n.$$

The weights $w_{m,0}$ in the first column of W (corresponding to the constant feature) are called the *biases* (sometimes also called *offsets*), since they are associated with the constant component of X and shift the preactivations.

The number of preactivations M, often called the *width* of the network, can be smaller or larger than the number of input features N. Neural networks should not be confused with dimensionality-reduction techniques such as principal component analysis. I will discuss how the width M is chosen in Section 7.

Step 2: Hidden units.

Next, the activation function ϕ is applied componentwise to the preactivations $z = \phi(a)$. Each resulting quantity $z_m = \phi(a_m)$, for $m = 1, \ldots, M$, is referred to as a *hidden unit*. In the literature, each hidden unit is also often called a *neuron*; the two terms are used interchangeably to denote the output of an activation function applied to a preactivation. The term *hidden* reflects the fact that these units are neither inputs nor outputs of the network; they are intermediate quantities.

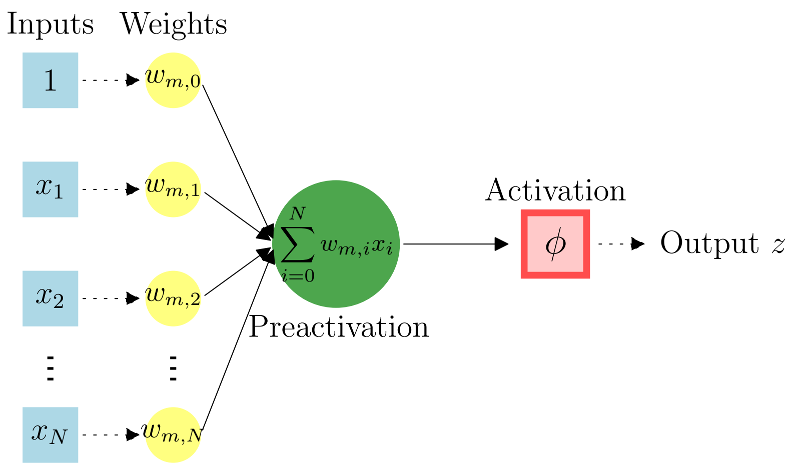

Figure 2. : A hidden unit

Figure 2 illustrates the construction of one hidden unit. The inputs (a constant for the bias and the N input features) are weighted, summed, and passed through the activation function to produce the unit's output. The activation function is the sole source of nonlinearity in the approximation; all other operations in the network (weighting, summation, and the linear combination of hidden units into the output) are linear.

Step 3: Linear output layer.

Finally, the hidden units are combined linearly to produce the neural network output:

$$\begin{aligned} g_\theta^{NN}(X) &= v_0 + v^\top z \\ &= v_0 + v^\top \phi(a) \\ &= v_0 + v^\top \phi(WX). \end{aligned}$$

An equivalent and more compact representation of a single-hidden-layer neural network is obtained by defining the affine output map:

$$r(z) \equiv v_0 + v^\top z.$$

With this notation, the network can be written as the composition:

$$g_\theta^{NN}(X) = (r \circ \phi \circ W)(X). \tag{8}$$

In Section 5, I extend this notation naturally to deep neural networks with multiple hidden layers.

1. Nested structure, representation learning, and encapsulated nonlinearity

The approximation implemented by a single-hidden-layer neural network has a nested structure. Steps 1 and 2 transform the input features X into a vector of hidden units $z \in \mathbb{R}^M$. The space $\mathbb{R}^M$ in which these objects live is usually referred to as the *representation space*.[17]

Informally, the hidden units z provide *informationally efficient* representations of the input features X, in the sense that a simple linear combination of them, as in Step 3, can approximate $f(X)$ accurately. This efficiency stems from the fact that neural networks encapsulate all nonlinear flexibility in a single component, the activation function $\phi(\cdot)$ applied in Step 2. Rather than working with complex nonlinear objects, such as the high-dimensional tensors that arise in Chebyshev polynomial approximations (recall the tensor in equation (7)), neural networks work with matrices and vectors outside $\phi(\cdot)$. As discussed in the subsection on "Classical Results" below, this structure is sufficiently expressive to approximate even highly intricate nonlinear functions and explains why neural networks can handle high-dimensional problems that would overwhelm polynomial-based methods.

The logic of finding informationally efficient representations is familiar to economists. In linear regression, undergraduate econometrics courses emphasize the importance of choosing a *good* representation of the regressors. For instance, it is common to work with logs or first differences rather than levels.

[17]In more general cases, the representation space might differ from $\mathbb{R}^M$, but I ignore that possibility for simplicity.

In the terminology of deep learning, such transformations correspond to manually *engineering* a representation of the input features, an approach known as *feature engineering*. This engineering exploits economic *domain knowledge* to make a linear model more informative.[18]

Such attempts to craft better representations by hand often rely on heuristics and can be brittle, leaving students frustrated by the lack of clear procedures for discovering useful representations beyond textbook examples. Neural networks replace this guesswork with a constructive procedure. Through training, W determines how the input features are translated and deformed so as to generate representations optimally suited to approximating $f(X)$.

However, the flexibility of this construction comes at a cost. The resulting network is highly parameterized: the collection of weights θ is high-dimensional, a fact that plays a central role in the discussion below of how neural networks are trained.[19]

Finally, the number of hidden units M and the choice of activation function $\phi(\cdot)$ define the *architecture* of the network, which I denote by $\mathcal{A}$. Section 7 discusses how this architecture is chosen. For now, I take $\mathcal{A}$ as given.

2. Activation functions

An effective activation function should be simple and computationally efficient to manipulate. Instead of designing sophisticated activation functions that can capture many shapes of the target function $f(\cdot)$, we want to have many yet easy-to-handle $\phi(\cdot)$'s.

The classical activation functions used since at least the early 1980s include (i) the identity function, $\phi(z) = z$ (which delivers a network that is equal to a linear regression; in other words, neural networks are a generalization of the basic idea of regression); (ii) a sigmoidal function, $\phi(z) = \frac{1}{1+e^{-z}}$; (iii) a step function: $\phi(z) = 1$ if $z > 0, \phi(z) = 0$ otherwise; and (iv) a hyperbolic tangent: $\phi(z) = \frac{e^{2z}-1}{e^{2z}+1}$. The hyperbolic tangent activation function is a very good default option for problems for which we know the target function we are looking for is smooth.[20]

More recently, the rectified linear unit (ReLU), $\phi(z) = \max(0, z)$, and its close relative, the softplus: $\phi(z) = \log(1+e^{z})$, have gained much popularity. Figure 3 plots these two activation functions. The ReLU is piecewise linear, equal to the 45^0 line for positive inputs and zero otherwise, making its derivative easy to compute (either zero or one. Furthermore, the non-differentiability at zero is usually not a concern: we can either ignore the points of non-differentiability in the final approximation or smooth them out by taking the mean of the left and right derivatives. If the function approximation must be everywhere differentiable and smoothing is undesirable, we can use the softplus activation function. The ReLU is a reasonable default option when little is known about the target function to approximate.

[18] *Feature engineering* involves transforming input features into a format more suitable for function approximation. Some transformations are simple, such as normalizing variables or taking first differences, whereas others, such as tokenizing text, can be more complex. See Zheng and Casari (2018) for an overview.

[19] Overparameterization, however, is not a drawback for neural networks when solving dynamic economic models in the way it would be in econometrics. I return to this issue later when discussing implicit regularization.

[20] See slide deck 4 in `www.sas.upenn.edu/~jesusfv/deeplearning.html` for a more complete discussion of activation functions.

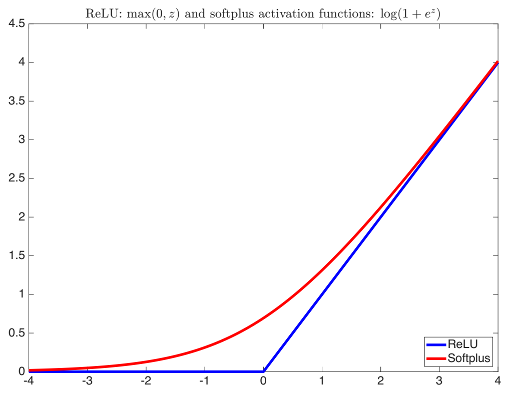

Figure 3. : ReLU and softplus activation functions

When the desired output is a probability vector over K categories (for example, when using a neural network to approximate a mixed strategy in game theory), we introduce an output layer that maps a vector of *logits* $s(X) \in \mathbb{R}^K$ into the probability simplex via the *softmax* function:

$$\pi_k(X) = \frac{\exp\bigl(s_k(X)\bigr)}{\sum_{j=1}^{K} \exp\bigl(s_j(X)\bigr)}, \qquad k = 1, \ldots, K,$$

so that $\pi(X)$ has strictly positive components that sum to one.

In a neural network, the logits are typically obtained as an affine transformation of the hidden layer (or, in the deep architectures in Section 5, of the last hidden layer). Extending our previous notation to accommodate vector-valued outputs, we write $s(X) = V_0 + V z(X)$, where $z(X) \in \mathbb{R}^M$ denotes the hidden-layer activations, $V_0 \in \mathbb{R}^K$ is the output bias vector, and $V \in \mathbb{R}^{K \times M}$ is the output weight matrix.

3. *Classical results*

Two classical results make neural networks, at least in theory, attractive as function approximators.

The first classical result is the universal approximation theorem by Cybenko (1989) and Hornik et al. (1989). This theorem states that the very simple network described by equation (8) can approximate any Borel-measurable function mapping finite-dimensional spaces to any desired degree of accuracy. This theorem ensures that, if we are willing to add sufficient hidden units to the network, we will be able to find a sufficiently good approximation of the target function $f(\cdot)$, even if this function is not differentiable or has jumps (see Kidger and Lyons, 2020, for a more general formulation of the theorem).

Instead of proving this result formally, a simple example illustrates the idea. First, note that, despite their simplicity, activation functions such as ReLU are highly flexible. The first two rows of Figure 4 plot nine ReLUs for a unidimensional input x with different weights $z = \phi(a) = \max(0, a) = \max(0, \theta_0 + \theta_1 x)$. In the top left panel, I draw the basic ReLU when $\theta_0 = 0$ and $\theta_1 = 1$. In the top center panel, I set $\theta_0 = 1$, which shifts the activation to the left. Conversely, in the top right panel, I set $\theta_0 = -1$, which shifts the activation to the right.

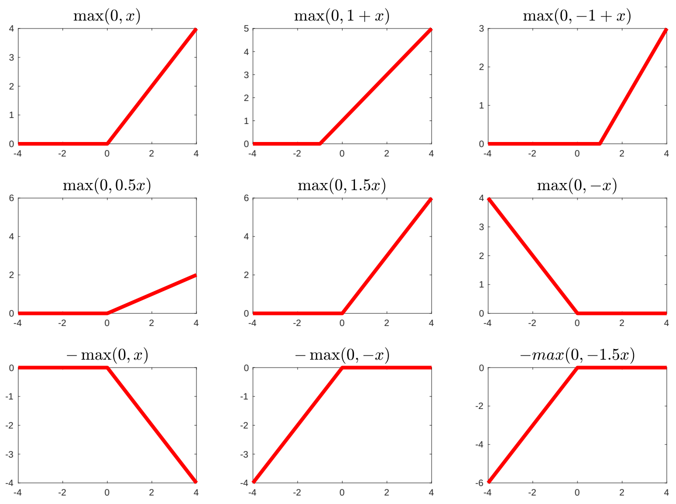

Figure 4. : ReLUs with different weights

In the center row, I play with θ_1. By making $\theta_1 < 1$, the linear part of the activation function becomes flatter (left panel). Conversely, by setting $\theta_1 > 1$, I make the activation function steeper (center panel). Finally, by changing the sign, $\theta_1 < 0$, I flip the activation function leftward (right panel)

In the bottom row of Figure 4, I change the weight in front of the unit z. Multiplying z by a negative number flips the ReLU downward, either to the right (left panel) or to the left (center panel), and also makes it steeper (right panel).

The careful reader might have noticed that different combinations of weights θ in the preactivation and output layer of the z's can deliver the same result (e.g., $-\max(0, -1.5x) = -1.5\max(0, -x)$). This is not a concern, since we do not care about the values of the θ's themselves but about the function approximation they deliver. If two different values of θ yield the same $g_\theta^{NN}(X)$, we are indifferent between them: it is the same function, merely written in two different but equivalent ways.

Thus, one should resist the temptation to draw an analogy with lack of identification in econometrics. Identification is problematic when the precise value of the parameters matters because they carry a sharp economic interpretation (e.g., the discount factor or risk aversion). In contrast, here the weights are not objects of interest; their specific values are irrelevant as long as they generate the same approximation.

Next, let me show how this flexibility can be put to good use. Imagine that we want to approximate the function $f(x) = x^3 + x^2 - x - 1$ in the interval $[-2, 1.5]$.[21] Despite its simplicity, this is an interesting function to approximate, as it has a global

[21] Here, I use x instead of X to make it clear that we are dealing with a one-dimensional approximation.

minimum at the left corner of its interval (-2), a global maximum at the right corner (1.5), an interior local maximum at -1, and an interior local minimum at 0.33.

If we use one ReLU unit to approximate this function,

$$g_\theta^1(x) = -\max(0, -7.7 - 5x),$$

we capture part of the shape of the function of interest, but we still miss much (for now, do not be concerned about how I determined the weights -7.7 and -5; I will return to it later). See the approximation (red line), plotted in the top left panel of Figure 5, against the target function (blue line).

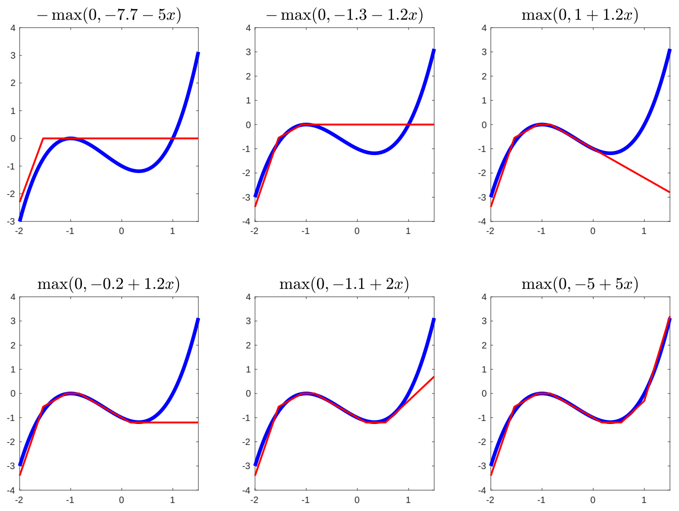

Figure 5. : A six ReLUs approximation

Let us add a second ReLU unit:

$$g_\theta^2(x) = g_\theta^1(x) - \max(0, -1.3 - 1.2x).$$

The new approximation, plotted in the top center panel of Figure 5, begins to capture additional features of $f(x)$ (using the same color convention as before). I have kept the weights of $g_\theta^1(x)$ unchanged. The optimal values of the θ's should vary as the number of units in the network's training increases; however, I keep them constant here to illustrate the procedure.

We continue iterating, adding more ReLU units. By the time we reach six units (bottom right panel), $g_\theta^6(x)$ closely approximates $f(x)$. Adding further units would bring us arbitrarily close to the target.

A convenient feature of the ReLU activation function is its local adaptability: because each unit is zero on one side of its kink, adding a new unit does not alter the approximation in the region where that unit is inactive. This allows local refinements without affecting other parts of the approximation, a property that is difficult to achieve with Chebyshev polynomials, which are nonzero almost everywhere.

As noted in Section 2, the curse of dimensionality in solving dynamic equilibrium models has two aspects. The first aspect concerns approximation. A classical insight due to Barron (1993) is that, for an important class of target functions known as the Barron class (roughly, functions with suitable spectral decay, a natural case for applications in economics), a single-hidden-layer network can achieve mean integrated squared error of order $O(1/M)$, where M denotes the number of hidden units.

Crucially, this rate does not depend on the dimension N of the input; for functions in the Barron class, neural networks break the approximation curse. This result does not extend to all target functions; even within the Barron class, the implicit constants in $O(1/M)$ can be large in economically relevant problems. However, more recent results show that deep networks can help circumvent these restrictions (Jacot et al., 2024).

To appreciate this advantage of neural networks, consider the alternative. For generic N-dimensional series methods (e.g., the Chebyshev polynomials in Section 3) and under standard smoothness assumptions, the corresponding rates often take the form $O(D^{-2s/N})$, where D is the number of basis terms and s measures the order of differentiability of the target (e.g., $s = 1$ for Lipschitz functions, $s = 2$ for functions with bounded second derivatives). The appearance of N in the exponent exposes the curse of dimensionality.

The second aspect of the curse of dimensionality concerns the availability of evaluation points. In very high-dimensional problems, whether evaluation points are obtained from a grid or from simulation, they will cover only a small fraction of the state space, even with powerful hardware. Finding the network weights, θ, then becomes difficult. There is not enough information for the network (or any numerical approximation method) to *learn* the shape of the target function $f(\cdot)$ and accurately fit $f(X)$ for points X outside the evaluation set. No approximation architecture can overcome this second curse because it hinges on the fundamental scarcity of information in high-dimensional state spaces. This second aspect can only be mitigated by efficient sampling, simulation design, or the exploitation of known structure in the problem.

5. *Deep neural networks*

Now, I introduce *deep neural networks*, the class of neural networks most commonly used in real-world applications. Since we already understand how a single-hidden-layer neural network operates, this extension is straightforward: we iteratively apply affine transformations followed by nonlinear activation functions.

Building on the representation introduced in Section 4, we now generalize it to multiple hidden layers. Concretely, we begin with the input features X and construct a first layer of hidden units. Let M^1 denote the number of hidden units in the first layer, and define

$$z^1 = \phi^1(W^1 X)\,,$$

where $W^1 \in \mathbb{R}^{M^1 \times (N+1)}$ is the matrix of weights for the first hidden layer (recall that X includes a constant), and $\phi^1(\cdot)$ is an activation function applied element by element. The superscript indicates the layer index, as this construction is repeated across multiple layers. As in the single-hidden-layer case, we implicitly augment the vector of hidden units in each subsequent layer with a constant equal to one, so that biases are incorporated through the corresponding weight matrices.

Instead of using the hidden units z^1 directly to form the approximation, as in a single-hidden-layer network, these outputs now serve as the inputs for the next layer. Let M^2 denote the number of hidden units in the second layer, and define

$$\begin{aligned} z^2 &= \phi^2\big(W^2 z^1\big) \\ &= \phi^2\big(W^2 \phi^1\big(W^1 X\big)\big) , \end{aligned}$$

where $W^2 \in \mathbb{R}^{M^2 \times (M^1+1)}$ is the corresponding weight matrix and $\phi^2(\cdot)$ is an activation function, possibly different from $\phi^1(\cdot)$. Both the number of hidden units and the choice of activation function may vary across layers.[22]

This pattern generalizes immediately. For a total of J hidden layers, and for $j = 1, \dots, J-1$, the hidden units in layer $j+1$ are given by

$$\begin{aligned} z^{j+1} &= \phi^{j+1}\big(W^{j+1} z^j\big) \\ &= \phi^{j+1}\big(W^{j+1} \phi^j\big(W^j \phi^{j-1}\big(W^{j-1} \cdots \phi^1\big(W^1 X\big) \cdots\big)\big)\big) , \end{aligned}$$

where $W^j \in \mathbb{R}^{M^j \times (M^{j-1}+1)}$ for $j = 2, \dots, J$.

Finally, the network output is obtained through an affine output from the last hidden layer:

$$\begin{aligned} y \approx g_\theta^{DL}(X) &= v_0 + v^\top z^J \\ &= v_0 + v^\top \phi^J\big(W^J z^{J-1}\big) \\ &= v_0 + v^\top \phi^J\big(W^J \phi^{J-1}\big(W^{J-1} \cdots \phi^1\big(W^1 X\big)\big) \cdots\big) . \end{aligned}$$

Defining, as in Section 4, the affine output layer

$$r(z) \equiv v_0 + v^\top z,$$

the deep neural network can be written compactly in compositional form as

$$g_\theta^{DL}(X) = \left(r \circ \phi^J \circ W^J \circ \cdots \circ \phi^1 \circ W^1\right)(X).$$

Taken together, these expressions describe the *forward propagation* mechanism of a deep neural network: a composition of affine maps and nonlinear activation functions, culminating in an affine output layer. All nonlinear flexibility of the approximation is contained in the activation functions $\phi^j(\cdot)$, while the remaining operations are affine and therefore computationally inexpensive and easy to parallelize.

When $J = 1$ (the case studied in Section 4), we are back to a one-hidden-layer network. When $J > 1$, the network is called *deep*, as it includes multiple hidden layers. Accordingly, *deep learning* refers to approximating an unknown function using a neural network with several layers.[23]

[22] From this point on, when z^j appears as an input to the next layer, it is implicitly augmented with a leading constant to account for the bias.

[23] In some parts of the literature, the term *deep* is reserved for networks with more than two or three hidden layers. Unfortunately, usage is not consistent across researchers.

Finally, the tuple

$$\theta = (v_0, v, W^1, W^2, \ldots, W^J)$$

collects the deep neural network weights, where $W^1 \in \mathbb{R}^{M^1 \times (N+1)}$, $W^j \in \mathbb{R}^{M^j \times (M^{j-1}+1)}$ for $j = 2, \ldots, J$, and $v \in \mathbb{R}^{M^J}$. As in the single-hidden-layer case, the "+1" accounts for the constant term that absorbs the bias at each layer.

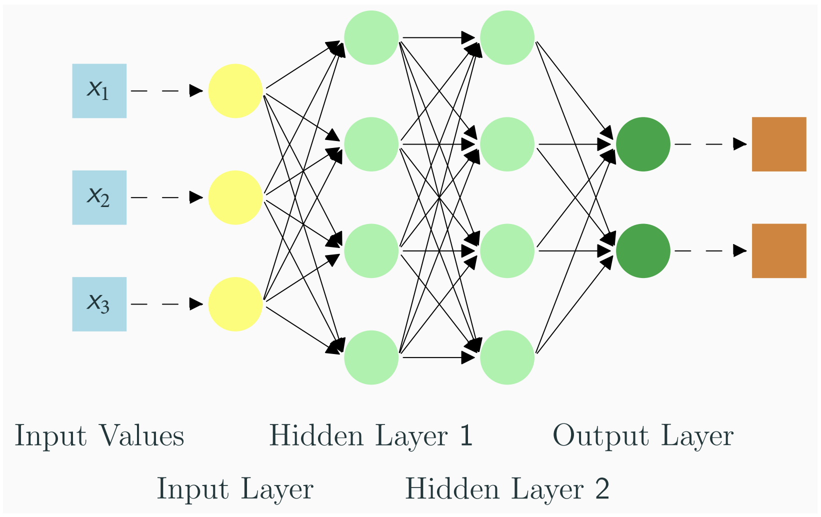

Figure 6. : A deep neural network

Figure 6 presents a graphical representation of a deep neural network with three input features (x_1, x_2, x_3) and two outputs. The circle units in each intermediate step are the hidden layers, whereas the final affine transformation constitutes the output layer. The figure depicts a network with two outputs (or *heads*), underscoring that neural networks can approximate vector-valued functions. This is easily achieved by treating the output weight vector v as a matrix rather than a vector.

Figure 6 also illustrates how the nested structure operates: the original inputs are transformed layer by layer into new hidden representations until we reach the final hidden units z^J, which are then combined linearly to form the approximation $g_\theta^{DL}(\cdot)$.

Deep neural networks often outperform shallow ones because they generate new representations of input features by recursively composing earlier ones. The central insight is that, by stacking many layers, each applying simple transformations, a deep neural network can efficiently approximate highly intricate functions. This nested structure enables the neural network to progressively build richer representations of the input features, which is the primary source of its expressive power.

Intuitively, the first layers typically capture simple patterns or correlations in the input features, while subsequent layers combine them into more complex ones. This hierarchy means that a deep neural network does not need to "learn everything at once," but can build up complicated behavior step by step.

For example, fewer hidden units in higher layers can summarize what lower layers have already extracted, thereby compressing learning into fewer features. In fact, these intermediate features are often interesting in their own right. Formally, with piecewise-linear activation functions (such as ReLU), one can prove that the neural network's expressiveness grows exponentially with its depth, while its computational cost increases only linearly (Goujon et al., 2024). For smooth nonlinear activations, similar efficiency gains from depth have been established in approximation theory, although without the same sharp exponential characterization (Poggio et al., 2017).

Deep neural networks also connect naturally with powerful mathematical tools. For instance, Ebrahimi Kahou et al. (2021) exploit the symmetric group to design deep neural networks for models with a finite but large number of heterogeneous agents. Unfortunately, space constraints prevent me from exploring these connections. Suffice it to say that deep learning opens the door for economists to use entire areas of mathematics that were previously of limited applicability in the discipline.

6. *Training the neural network*

So far, I have avoided any discussion of how the weights θ of the network are chosen. Determining these weights so that $g_\theta^{NN}(X)$ provides a good approximation to the target function $f(X)$ is known as *training* the network.

Training is performed by specifying a *loss function* that measures the quality of the approximation at the evaluation points $\{X_l\}_{l=1}^{L}$. For any evaluation point X_l, we define a *per-evaluation-point loss* $\ell_l(\theta)$. Natural choices for $\ell_l(\theta)$ when solving dynamic economic models include the Bellman equation residual, the Euler equation residual, the conditional expectation residual, and similar residuals arising from the model's equilibrium conditions.

While the specific form of the residual varies across applications, all cases share a common structure. Let $\mathcal{R}[g_\theta](X)$ denote the residual of a functional equation evaluated at state X when the unknown function f is approximated by a candidate solution g_θ parameterized by weights θ. For instance, in the stochastic neoclassical growth model of Section 2.4, we can use either the Bellman residuals, $\mathcal{R}[\widehat{V}_\theta](X_t)$, or the Euler equation/conditional expectation residual, $\mathcal{R}[\widehat{d}_\theta](X_t)$, where $\widehat{V}_\theta(X_t)$ and $\widehat{d}_\theta(X_t)$ are neural networks.

The per-evaluation-point loss is then the squared residual:

$$\ell_l(\theta) = \frac{1}{2}\mathcal{R}[g_\theta^{NN}](X_l)^2.$$

Given evaluation points $\{X_l\}_{l=1}^{L}$, the objective is the *training loss*:

$$\mathcal{L}_L(\theta) = \frac{1}{L}\sum_{l=1}^{L}\ell_l(\theta),$$

and training the neural network consists of solving

$$\theta^* = \arg\min_\theta \mathcal{L}_L(\theta). \tag{9}$$

The loss $\mathcal{L}_L(\theta)$, computed at finitely many L evaluation points, approximates a *population loss* that averages the residual over all states X the economy may visit. In principle, we would like to find θ such that the approximation error is small across all relevant states, not just at the evaluation points. However, specifying the distribution underlying the population expectation is nontrivial in economic applications, a point I develop in Section 8.

To assess whether the trained network generalizes beyond the training points, the researcher typically generates a second set of points to assess the solution. If the original evaluation points $\{X_l\}_{l=1}^L$ are the *training set* used to solve problem (9), the second set $\{X_p\}_{p=1}^P$, called the *test set* (or *validation set*), is used to evaluate the quality of the approximation on points not seen during training.

1. *Gradient descent and its variants*

We need an optimization algorithm that can handle the high dimensionality of θ, which can easily exceed hundreds of thousands of weights. Because of this requirement, most optimization algorithms for training neural networks are variants of gradient descent. Interestingly, the computational cost of gradient-based algorithms does not scale exponentially with $\dim(\theta)$. In fact, for problem (9), the loss landscape often becomes smoother when $\dim(\theta) \gg L$, so that gradient-based optimization is faster and more reliable than when $\dim(\theta) \approx L$. See Mei et al. (2018), Soudry et al. (2018), and Du et al. (2019) for details.[24]

The basic gradient descent algorithm generates iterates $\theta^{(k)}$ from an initialization $\theta^{(0)}$ according to the recursion

$$\theta^{(k+1)} = \theta^{(k)} - \eta^{(k)} \nabla_\theta \mathcal{L}_L\left(\theta^{(k)}\right), \tag{10}$$

using the (negative) gradient of the loss function, $-\nabla_\theta \mathcal{L}_L(\theta^{(k)})$, which points in the direction of steepest descent, and a learning rate $\eta^{(k)} > 0$. This learning rate controls the magnitude of each update. If $\eta^{(k)}$ is too small, convergence in equation (10) is slow and the algorithm may stall in flat regions of the loss surface. If $\eta^{(k)}$ is too large, the algorithm may overshoot the minimum or even diverge. The initialization $\theta^{(0)}$ can be chosen at random or based on domain knowledge.

The primary advantage of gradient descent is that it does not require evaluating the Hessian, unlike second-order methods such as Newton's method. This is crucial because computing and storing a Hessian is prohibitively costly when θ is high-dimensional.

Even without second-order information, computing the full gradient $\nabla_\theta \mathcal{L}_L(\theta)$ can be expensive when L is large. Repeatedly accessing the entire $\{X_l\}_{l=1}^L$ may overwhelm memory bandwidth and significantly slow computation.

To address this problem, practitioners rely on a simple yet powerful idea: instead of evaluating the gradient on the full training set $\{X_l\}_{l=1}^L$, the gradient is approximated using a single observation or a small random subset of $\{X_l\}_{l=1}^L$, which changes at each iteration. When a single evaluation point is used, the method is called *stochastic*

[24]See slide deck 4 in `www.sas.upenn.edu/~jesusfv/deeplearning.html` for more on optimization methods. Non-derivative-based optimization algorithms (such as Nelder–Mead, simulated annealing, or genetic algorithms) are generally far too slow to train a network because they require evaluating the objective function many times without exploiting gradient information.

gradient descent (SGD) (Robbins and Monro, 1951). When a small subset is used, the method is called *minibatch stochastic gradient descent*. In practice, however, researchers often use the term SGD to refer to both cases.

The intuition behind SGD and minibatch methods is that most of the information required to update θ (that is, to move in a direction close to steepest descent for problem (9)) is already contained in a small number of observations. Under standard sampling assumptions, a random subset of the training set provides an unbiased estimate of the full gradient.

While the resulting update is noisy, it is typically sufficiently accurate to approximate the full gradient and enable rapid progress. Employing the entire training set at every iteration is, therefore, often an inefficient use of computational resources.[25]

This strategy allows much faster training and facilitates robustness checks. For example, one can initialize the algorithm at different values of $\theta^{(0)}$ and check whether it converges to the same solution. Moreover, the randomness induced by stochastic gradient estimates injects a small but beneficial amount of noise into the optimization process. This noise can help the algorithm escape shallow local minima or extended flat regions of the loss surface, often improving generalization.[26]

In recent years, more sophisticated variants of gradient descent, such as ADAM (Kingma and Ba, 2017), have gained widespread adoption. These methods incorporate *momentum* and adaptive learning rates. Momentum relies on an exponential moving average of past gradients, which accelerates progress across flat regions of the loss surface and dampens oscillations in steep or noisy directions. ADAM further rescales gradients on a weight-by-weight basis, allowing each component of θ to have its own effective step size. This makes ADAM particularly well-suited to high-dimensional problems and to sparse gradients. For an accessible introduction to these optimization algorithms, see Aggarwal (2020).

2. *Backpropagation*

For computing the gradient $\nabla_\theta \mathcal{L}_L(\theta)$, we use *backpropagation*, which efficiently applies the chain rule to determine how the loss function changes as we vary the network's weights (Rumelhart et al., 1986).[27]

To illustrate the logic, consider a single-hidden-layer neural network and a loss $\ell_l(\theta) = \frac{1}{2}\mathcal{R}[g_\theta^{NN}](X_l)^2$ at one evaluation point X_l. By the chain rule:

$$\frac{\partial \ell_l(\theta)}{\partial \theta_i} = \mathcal{R}[g_\theta^{NN}](X_l) \cdot \frac{\partial \mathcal{R}[g_\theta^{NN}](X_l)}{\partial \theta_i}. \tag{11}$$

The term $\frac{\partial \mathcal{R}}{\partial \theta_i}$ depends on the structure of the residual. For functional residuals such as Bellman or Euler equations, where both the candidate solution and the continuation value depend on θ, the derivative includes contributions from each occurrence of θ in

[25] In modern hardware, particularly graphical processing units, the main bottleneck in training neural networks is not floating-point computation but memory access. SGD and minibatch methods are designed to operate efficiently under these memory constraints.

[26] A common practical strategy is to use SGD or minibatch methods to approach a neighborhood of a minimum and then switch to full-batch gradient descent for final convergence.

[27] Versions of backpropagation appeared earlier in the control theory and applied mathematics literature, but Rumelhart et al. (1986) popularized the method and sparked modern interest in neural networks.

the residual.

Backpropagation computes these derivatives by propagating errors backward through the network's layers. The reuse of intermediate quantities makes backpropagation computationally efficient. Without it, each partial derivative would need to be computed independently, an approach that scales linearly with the number of weights and quickly becomes infeasible for large neural networks.[28] With ReLU activations, moreover, $\phi'(a)$ is particularly simple to evaluate.[29]

Finally, modern open-source machine learning libraries such as `PyTorch` and `JAX` rely on automatic differentiation, which dovetails seamlessly with backpropagation and delivers all required derivatives at high speed, even for very large neural networks.

3. Regularization

Regularization refers to techniques that constrain the optimization problem (9) to prevent overfitting on the training points and improve performance on the test set. The concern that regularization addresses is that a network with many weights can fit the training data very well while performing poorly on new test data. This tradeoff is familiar to econometricians as the bias–variance tradeoff: a regression with many regressors can achieve a high R^2 on the estimation sample while predicting poorly on a held-out sample (Hastie et al., 2009). *Generalization* refers to the ability of a trained network to perform well on such new points, typically assessed on the test set.

Regularization can be achieved explicitly or implicitly.

Explicit regularization

To explicitly regularize problem (9), we can augment the objective function with penalty terms that discourage overly large weights (often excluding bias terms). The intuition is that large weights allow the network to fit idiosyncratic features of the training points, leading to poor generalization. By penalizing large weights, we encourage smoother, more stable approximations.

Common choices include:

1) A LASSO penalty (absolute-value norm):

$$\lambda\|\theta\|_1, \text{ where } \lambda > 0 \text{ and } \|\theta\|_1 \equiv \sum_i |\theta_i|,$$

which induces sparsity by forcing some weights to exactly zero.

2) A ridge penalty (squared Euclidean norm, also known as Tikhonov regularization):

$$\lambda\|\theta\|_2^2, \text{ where } \lambda > 0 \text{ and } \|\theta\|_2^2 \equiv \sum_i \theta_i^2,$$

[28]Backpropagation becomes even more critical in deep neural networks, where it allows gradients to be propagated backward through multiple layers with only modest additional computation.

[29]The ReLU is not differentiable at zero. In practice, gradient-based methods proceed using subgradients, and the set of nondifferentiable inputs has Lebesgue measure zero under continuous feature distributions. This is standard in nonsmooth optimization and is handled routinely by automatic differentiation frameworks.

which shrinks weights toward zero without inducing exact sparsity.

3) A combination of both penalties (elastic net):

$$\lambda_1\|\theta\|_1 + \lambda_2\|\theta\|_2^2, \text{ where } \lambda_1, \lambda_2 > 0.$$

These penalty terms have the advantage of being transparent and interpretable, but they require the specification of additional hyperparameters (λ or λ_1 and λ_2), which must be tuned (see my discussion in Section 7 about architecture design).

Implicit regularization.

Interestingly, SGD and its variants exhibit implicit regularization: they bias θ^* toward weights that generalize well without explicit penalty terms. In particular, a network with sufficient width and depth for solving dynamic economic models will exceed the *interpolation threshold*, the point at which the number of parameters is large enough to fit the training data exactly. Beyond this threshold, the optimization problem in (9) becomes indeterminate, as many different values of θ satisfy $\mathcal{L}_L(\theta) = 0$. Which θ does the optimizer select?

It turns out that high-dimensional gradient-based optimizers converge to a θ^* that is a min-norm solution (Wilson, 2025), that is,

$$\theta^* \equiv \arg\min_{\theta:\, g_\theta^{NN} \in \mathcal{A}} \|g_\theta^{NN}\|_\psi \tag{12}$$

$$\text{s.t. } \mathcal{L}_L(\theta) = 0. \tag{13}$$

In words: among all the possible solutions that interpolate the data perfectly, gradient-based optimizers pick the $g_{\theta^*}^{NN}$ that, within the predetermined architecture $\mathcal{A}$, minimizes the functional seminorm $\|\cdot\|_\psi$ (which seminorm depends on each model and architecture). For instance, $\|\cdot\|_\psi$ may be a Sobolev seminorm measuring the integral of squared derivatives: flat functions have small seminorms, wiggly functions have large ones.[30]

Figure 7 illustrates implicit regularization. The top-left panel shows 12 training points (red) sampled from $Y = 2(1-e^{-|X+\sin(X^2)|})$. With 31 weights (top-right panel), a single-hidden-layer neural network with ReLU activation functions underfits. At 2,401 weights (bottom-left panel), it interpolates but is wiggly. At 12,001 weights (bottom-right panel), it still interpolates but is flatter: the implicit regularization selects the smoother solution.

Ebrahimi Kahou et al. (2024) prove that, for a broad class of equilibrium models, flatter solutions have two advantages. First, they satisfy the transversality condition. Second, they generalize better. In practice, most papers that use deep learning to solve dynamic economic models rely solely on implicit regularization, without explicit penalty terms. However, understanding explicit regularization remains valuable, as some applications may benefit from it.[31]

[30] I work with seminorms rather than norms to allow functions different from the zero function to have $\|g_\theta\|_\psi = 0$; for instance, a function that is constant over its domain but not equal to zero.

[31] See slide decks 8 and 9 in `www.sas.upenn.edu/~jesusfv/deeplearning.html` for more on implicit regularization and related ideas.

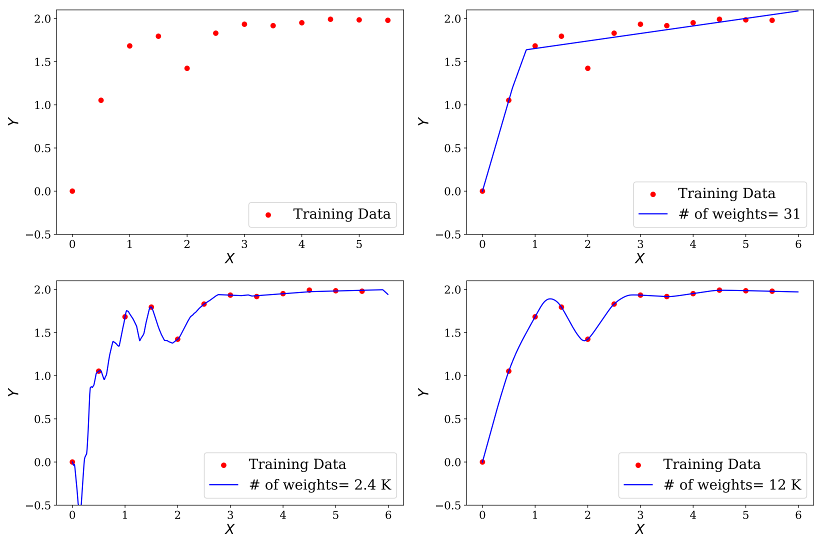

Figure 7. : Interpolation with an overparameterized neural network.

Having described how a network is trained, I now turn to the choices that must be made before training begins: the network architecture.

7. *Architecture design*

In previous sections, I have taken many aspects of the network architecture $\mathcal{A}$ as given. In practice, however, these choices must be made explicitly (hence the importance of access to a robust machine learning library, which automates or at least accelerates many of these design steps).

More concretely, one needs to choose a number of *hyperparameters*, including (i) the number of hidden layers; (ii) the activation function used in each layer; (iii) the number of hidden units per layer; and, on the training side, (iv) the optimization algorithm, the learning-rate schedule, the number of epochs and the minibatch size, and the weights on explicit regularization penalty terms (if applicable).

A few considerations are useful. First, the design of a neural network architecture is not very different in content or spirit from that of any other function approximator. For example, when implementing value function iteration, one must choose a grid for the state variables, an optimization method, and a convergence criterion. In this sense, neural networks are neither easier nor harder to design than other numerical methods. Second, nothing substitutes for experience in architecture design. Coding relies on established procedures, but it is also an art. Third, all design procedures are guided by the loss function $\mathcal{L}_L(\theta)$.

Let me discuss three of these procedures. The first is to introduce an explicit regularization term and iterate over selected architectural dimensions while holding other hyperparameters fixed. For example, one might start with $J = 3$ hidden layers and examine how the loss on the validation set changes as J is reduced to 2 or increased to 4. Does the validation loss decrease or increase, and by how much relative to the change induced by the regularization penalty?

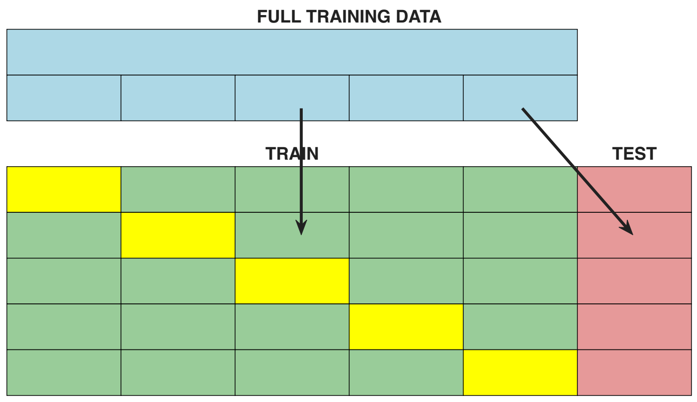

Figure 8. : Cross-validation

The second procedure is cross-validation, as illustrated by Figure 8. The training set (first row) is split into multiple subsets (second row), commonly referred to as folds. In this example, five folds are used. The method for dividing the data into folds should account for the problem's specific characteristics. For instance, if the data exhibit temporal dependence, it is crucial to preserve this structure, as in approaches such as block bootstrapping for time series.

The network is then trained on combinations of folds, with one fold reserved for validation. The first row in the bottom part of Figure 8 trains the network using folds two to five and validates it on fold one. Each subsequent row follows the same process, rotating the fold designated for validating.

The goal is to fine-tune the architecture's hyperparameters to minimize validation error across the different folds and ensure robust generalization, that is, the stability of performance under small perturbations of the training data. In that sense, cross-validation in deep learning resembles the logic of model selection in statistics (Arlot and Celisse, 2010). Many refinements of this basic cross-validation procedure are available.

The third regularization technique is dropout, illustrated in Figure 9. The left panel shows a fully connected network with three inputs, two hidden layers, and two outputs. During training, dropout randomly sets a fraction of hidden-unit activations to zero (say, 50%), effectively thinning the network, as depicted schematically in the right panel. Since units cannot rely on specific neighbors being active, the network is pushed to learn more redundant representations. If the training loss increases sharply when dropout is introduced, the model may benefit from a different regularization, additional capacity, or longer training. If the loss remains broadly stable, the architecture likely has sufficient capacity, and the resulting representation is relatively robust.

So far, I have presented basic feedforward neural networks, both shallow and deep. However, a menagerie of alternative architectures has been engineered to exploit

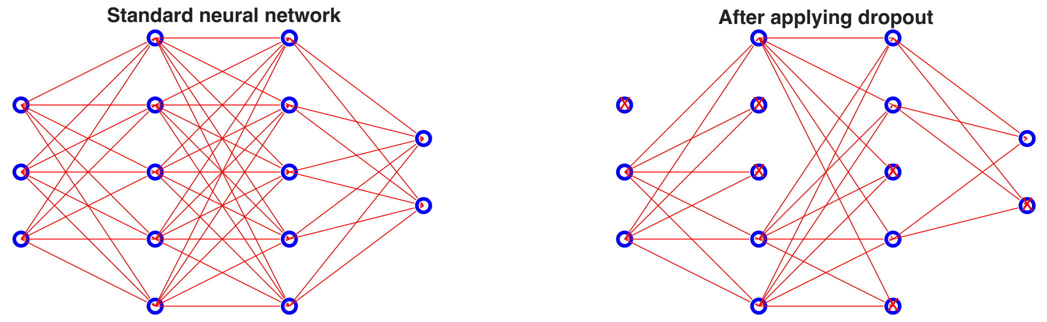

Figure 9. : Dropout

specific structures or inductive biases inherent to particular problems. See Prince (2023) for a textbook introduction to the most common architectures and slide deck 6 in `www.sas.upenn.edu/~jesusfv/deeplearning.html`.

Since these alternative architectures are less common in the solution of dynamic economic models, I omit discussion of them for brevity.

8. *Applying deep learning to solve functional equations*

The previous sections introduced the core ideas in function approximation for solving dynamic economic problems (residual minimization, neural network training, and sampling), but always in simplified settings. I now present a general framework for dynamic programming that accommodates the constraints encountered in practice, including occasionally binding constraints, equality restrictions, and the associated Lagrange multipliers. This generality requires heavier notation, but it nests most dynamic programming problems in macroeconomics, finance, industrial organization, and related fields as special cases.

I develop the framework in both discrete and continuous time. The choice of formulation depends on the context and is a decision for the researcher.

After this section, the reader should be able to adapt the approach to problems formulated in terms of optimality conditions (or other equilibrium conditions) rather than value functions without much difficulty.

1. *A general framework*

In discrete time, our goal is to solve the Bellman equation of a decision maker:

$$V(X) = \max_{\alpha} r(X, \alpha) + \beta \mathbb{E} V(X'), \tag{14}$$

where $V(\cdot)$ is the value function, which depends on N states X, $r(\cdot,\cdot)$ is the return function of the agent, which depends on the states and the P controls α, β is the discount factor, and $\mathbb{E}$ is the conditional expectation operator. Also, I am following the recursive notation introduced in Section 2.

Equation (14) is subject to three constraints:

$$X' = f(X, \alpha, \varepsilon) \tag{15}$$
$$G(X, \alpha) \leq \mathbf{0} \tag{16}$$
$$H(X, \alpha) = \mathbf{0}. \tag{17}$$

Constraint (15) describes the law of motion for the states, linking current states and controls to next-period states and shocks ε with respect to which we take the conditional expectation in equation (14). Constraint (16) is an inequality constraint in L_1 dimensions, requiring a function of states and controls to be (weakly) negative. For example, it could represent an occasionally binding constraint, such as a participation or liquidity constraint. Constraint (17) is an equality constraint in L_2 dimensions, forcing a function of states and controls to be exactly zero. This framework is highly general: constraints are often absent or incorporated into the Bellman equation.

If we reformulate the problem in continuous time, we have a Hamilton-Jacobi-Bellman (HJB) equation:

$$\rho V(X) = \max_{\alpha} r(X, \alpha) + \nabla_X V(X)^\top f(X, \alpha) + \frac{1}{2} \mathrm{tr}\left(\sigma(X)^\top \Delta_X V(X) \sigma(X)\right),$$

subject to the same inequality and equality constraints as before:

$$G(X, \alpha) \leq \mathbf{0} \tag{18}$$
$$H(X, \alpha) = \mathbf{0}, \tag{19}$$

where $\nabla_X V(X)$ is the gradient of the value function, $\Delta_X V(X)$ is the Hessian, and $\sigma(X)$ is the diffusion matrix governing the stochastic evolution of the states.[32]

The law of motion $X' = f(X, \alpha, \varepsilon)$ in discrete time now appears directly in the HJB equation: $f(X, \alpha)$ is the drift and $\sigma(X)$ captures uncertainty. Itô's lemma has allowed me to eliminate the expectation operator, which is always a costly step to approximate in any numerical procedure.[33]

To tackle this problem with deep learning, let me define four neural networks:

1) $\widehat{V}_{\theta^V}(X) : \mathbb{R}^N \rightarrow \mathbb{R}$ to approximate the value function $V(X)$.
2) $\widehat{\alpha}_{\theta^\alpha}(X) : \mathbb{R}^N \rightarrow \mathbb{R}^P$ to approximate the decision function α.
3) $\widehat{\mu}_{\theta^\mu}(X) : \mathbb{R}^N \rightarrow \mathbb{R}^{L_1}$ to approximate the multiplier μ associated with the inequality constraint.
4) $\widehat{\lambda}_{\theta^\lambda}(X) : \mathbb{R}^N \rightarrow \mathbb{R}^{L_2}$ to approximate the multiplier λ associated with the equality constraint.

Since training these four neural networks simultaneously is equivalent to training a neural network with the same four outputs, I will refer to one neural network from now on, and I can accumulate all weights in the tuple $\theta = (\theta^V, \theta^\alpha, \theta^\mu, \theta^\lambda)$.

[32] A more general case would let α affect the volatility, e.g., $\sigma(X, \alpha)$, at the cost of heavier notation below.
[33] Evaluating integrals is subject to its own curse of dimensionality.

2. The loss function

Next, we define a loss function. To that end, we will define several error criteria. First, in discrete time, we will have the Bellman residual:

$$\mathrm{err}_{B,\theta}(X) \equiv r(X, \widehat{\alpha}_{\theta^\alpha}(X)) + \beta \mathbb{E} \widehat{V}_{\theta^V}(X') - \widehat{V}_{\theta^V}(X),$$

and the policy function residual:

$$\begin{aligned}\mathrm{err}_{\alpha,\theta}(X) \equiv \frac{\partial r(X, \widehat{\alpha}_{\theta^\alpha}(X))}{\partial \alpha} &+ \beta \mathbb{E}\left[D_\alpha f(X, \widehat{\alpha}_{\theta^\alpha}(X), \varepsilon)^\top \nabla_X \widehat{V}_{\theta^V}(X')\right] \\ &- D_\alpha G(X, \widehat{\alpha}_{\theta^\alpha}(X))^\top \widehat{\mu}_{\theta^\mu}(X) - D_\alpha H(X, \widehat{\alpha}_{\theta^\alpha}(X))^\top \widehat{\lambda}_{\theta^\lambda}(X),\end{aligned}$$

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

The equivalent errors in continuous time are the HJB residual:

$$\begin{aligned}\mathrm{err}_{HJB,\theta}(X) \equiv\ & r(X, \widehat{\alpha}_{\theta^\alpha}(X)) + \nabla_X \widehat{V}_{\theta^V}(X)^\top f(X, \widehat{\alpha}_{\theta^\alpha}(X)) \\ &+ \frac{1}{2}\mathrm{tr}\left(\sigma(X)^\top \Delta_X \widehat{V}_{\theta^V}(X) \sigma(X)\right) - \rho \widehat{V}_{\theta^V}(X)\end{aligned}$$

and the policy function residual:

$$\begin{aligned}\mathrm{err}_{\alpha,\theta}(X) \equiv \frac{\partial r(X, \widehat{\alpha}_{\theta^\alpha}(X))}{\partial \alpha} &+ D_\alpha f(X, \widehat{\alpha}_{\theta^\alpha}(X))^\top \nabla_X \widehat{V}_{\theta^V}(X) \\ &- D_\alpha G(X, \widehat{\alpha}_{\theta^\alpha}(X))^\top \widehat{\mu}_{\theta^\mu}(X) - D_\alpha H(X, \widehat{\alpha}_{\theta^\alpha}(X))^\top \widehat{\lambda}_{\theta^\lambda}(X),\end{aligned}$$

with the same notation for the submatrices as above.

Second, we will have the constraint errors (which are the same for discrete and continuous time). The constraint error is itself composed of the primal feasibility residual:

$$\begin{aligned}\mathrm{err}_{PF_1,\theta}(X) &\equiv \max\{0, G(X, \widehat{\alpha}_{\theta^\alpha}(X))\} \\ \mathrm{err}_{PF_2,\theta}(X) &\equiv H(X, \widehat{\alpha}_{\theta^\alpha}(X)),\end{aligned}$$

the dual feasibility residual:

$$\mathrm{err}_{DF,\theta}(X) = \max\{0, -\widehat{\mu}_{\theta^\mu}(X)\},$$

and the complementary slackness residual:

$$\mathrm{err}_{CS,\theta}(X) = \widehat{\mu}_{\theta^\mu}(X)^\top G(X, \widehat{\alpha}_{\theta^\alpha}(X)).$$

This last error may involve component-wise penalization or alternative L_1-slackness formulations that prevent cross-cancellation, in which a violation in one constraint offsets a large violation in another.

We aggregate these errors at one evaluation point X_l using a squared-error loss:

$$\ell_l(\theta) = \omega_1 \left\| \mathrm{err}_{B/HJB,\theta}(X_l) \right\|_2^2 + \omega_2 \left\| \mathrm{err}_{\alpha,\theta}(X_l) \right\|_2^2 + \omega_3 \left\| \mathrm{err}_{PF_1,\theta}(X_l) \right\|_2^2 + \omega_4 \left\| \mathrm{err}_{PF_2,\theta}(X_l) \right\|_2^2 + \omega_5 \left\| \mathrm{err}_{DF,\theta}(X_l) \right\|_2^2 + \omega_6 \left\| \mathrm{err}_{CS,\theta}(X_l) \right\|_2^2,$$

for a collection of weights ω_i, $i = 1, \ldots, 6$. Because equilibrium conditions may be expressed in different units (for example, consumption in levels and interest rates in percentage points), a natural baseline is to normalize each residual so that a given percentage deviation carries comparable weight across equations, and then set the ω_i equal.[34]

Two points are worth highlighting. First, some of these errors may disappear if certain constraints are relaxed. Second, I solve jointly for the value function and the decision rule. While, in principle, one can recover the latter from the former (and conversely), practical experience suggests that solving for both simultaneously yields more accurate results (Fujimoto et al., 2022).

3. *The equilibrium loop revisited*

Recall that in Subsection 2.5, I argued that the natural goal would be to solve:

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{\xi^*(\cdot)} \ell_l(\theta),$$

where the expectation is taken with respect to the ergodic distribution $\xi^*(\cdot)$ of the states X, but that since ξ^* depends on the solution itself (the *equilibrium loop*), we instead solve:

$$\theta^* = \arg\min_{\theta} \frac{1}{L} \sum_{l=1}^{L} \ell_l(\theta).$$

We can apply here the iterative approach outlined in Subsection 2.5: guess an initial θ^0, generate evaluation points by simulating from the model using the decision rule implied by θ^0, retrain the network to obtain θ^1, regenerate evaluation points, and iterate until convergence.

For the initial guess θ^0, we can solve the dynamic programming problem using a first-order perturbation while ignoring constraints (16) and (17), which is easy to implement with standard software, and use the (misspecified) solution to generate an initial epoch and train the network. Alternatively, one could solve the model using value function iteration (in discrete time) or finite differences (in continuous time) on a very coarse grid (which yields a very fast solution), and then proceed as above. While not entirely accurate, this preliminary solution is a strong initial candidate.

The convergence properties of this algorithm were discussed in Subsection 2.5; the practical strategies for the initial guess described above are designed to address them.

[34]Normalization can be done, for instance, by the steady-state value or by the sample standard deviation of each residual. More elaborate adaptive weighting schemes have been proposed in the machine learning literature on multi-task learning (e.g., Kendall et al., 2018), which could be fruitfully adapted to dynamic equilibrium models. In any case, the researcher should confirm that reasonable variations in the weights do not materially alter the computed solution.

Variations of this algorithm were pioneered by Fernández-Villaverde et al. (2020), Maliar et al. (2021) and Azinovic et al. (2022) and applied by Gorodnichenko et al. (2020), Gopalakrishna (2021), Sauzet (2021), Huang (2023), Azinovic and Zemlicka (2024), Duarte et al. (2024), Nuño et al. (2024), Folini et al. (2024), and Villa and Valaitis (2024).

9. *An example: The continuous-time neoclassical growth model*

I illustrate the previous steps using the continuous-time deterministic neoclassical growth model. Since this is a one-dimensional problem, there are much faster ways to solve this model than using a deep neural network (e.g., finite differences, e.g., Candler, 2001), and one would rather reserve deep learning for more complex dynamic programming problems. But, from a pedagogical perspective, this is a great example due to its simplicity.

A social planner is solving:

$$\max_c \int_{t=0}^{\infty} e^{-\rho t} u(c)\, dt$$

where $u(c) = \frac{c^{1-\gamma}}{1-\gamma}$, subject to $\dot{k} = Ak^\alpha - \delta k - c$ and $k > 0$ (for simplicity, I am dropping the t index in the variables).

This problem can be formulated as an HJB equation in terms of a value function on the state variable $k > 0$:

$$\rho V(k) = \max_c \{u(c) + V'(k)[Ak^\alpha - \delta k - c]\}.$$

To solve this problem, we approximate a value function $\widehat{V}_{\theta^V}(k)$ and a decision rule for consumption $\widehat{c}_{\theta^c}(k)$ using a deep neural network with two outputs.[35]

To train this network, we define an HJB residual:

$$\text{err}_{HJB,\theta}(k) = \rho \widehat{V}_{\theta^V}(k) - u\left(\widehat{c}_{\theta^c}(k)\right) - \frac{\partial \widehat{V}_{\theta^V}(k)}{\partial k}\left[Ak^\alpha - \delta k - \widehat{c}_{\theta^c}(k)\right],$$

and a decision function residual (in units of consumption):

$$\text{err}_{c,\theta}(k) = (u')^{-1}\left(\frac{\partial \widehat{V}_{\theta^V}(k)}{\partial k}\right) - \widehat{c}_{\theta^c}(k).$$

Thus, our per-evaluation-point loss function is $\ell_l(\theta) = \left(\text{err}_{HJB,\theta}(k)\right)^2 + \left(\text{err}_{c,\theta}(k)\right)^2$, assuming equal weights for simplicity.

Designing the details of the implementation

Let me discuss the details of the numerical implementation:

[35]For clarity in the exposition, I am not considering the non-negativity constraint in capital. In the code, I impose that capital cannot be below a small positive value.

1) For structural parameters, I adopt conventional values: $\gamma = 2$, $\rho = 0.04$, $A = 0.5$, $\alpha = 0.36$, and $\delta = 0.05$. These values imply $k_{ss} = 2.95$.

   To pick structural parameter values, the researcher should proceed as in any other calibration exercise (e.g., matching relevant empirical moments, borrowing estimates from the micro literature, etc.). One could also embed the solution method within an estimation procedure (see Fernández-Villaverde et al., 2016, for how to combine solution and structural estimation in equilibrium models).

2) I select an architecture with three layers, each containing eight hidden units. The choice of layers and hidden units follows the guidelines in Section 7. In particular, I monitored how the residual error evolved as I varied the architecture. Three layers, each with eight hidden units, delivered high accuracy. Often, it is preferable to use more layers with fewer hidden units per layer than fewer layers with more hidden units per layer.

3) I use $\tanh(x)$ as the activation function in all layers. Since the problem is smooth and concave, $\tanh(x)$ usually performs better than ReLU. When the problem's properties are less clear, ReLU is often a safer default, although it may require more layers and hidden units to match the same accuracy.

4) In the code, I clamp the capital input to lie within $[0.1, 10]$, ensuring that the network is never evaluated at economically meaningless values (e.g., capital less than 3% or more than 339% of k_{ss}).

5) Implicit regularization ensures that the transversality condition holds in this application without explicitly checking for it (Ebrahimi Kahou et al., 2024). In other cases, the researcher can examine long-horizon simulations from different initial states to ensure that the system does not violate transversality.

6) I use 100 evaluation points in the state space for computing the residual errors. This is sufficiently large that the loss function provides a reliable signal for optimization; I verified that varying the number does not affect the solution.

7) At the start of each epoch, I draw the 100 evaluation points afresh from a normal distribution centered at the steady-state value of capital, with a standard deviation of $k_{ss}/3$. The choice of variance is a tuning parameter. In this application, a standard deviation of $k_{ss}/3$ is wide enough that the evaluation points span enough of the range of capital to get good accuracy within $[0.1, 10]$. I verify that results are robust to moderate changes in this choice.

   For stochastic models with richer state spaces, my preferred approach is to compute a first-order perturbation of the model (perhaps ignoring constraints or complications) and use simulated paths as evaluation points; the perturbation solution provides a rough guide to the region of the state space that the economy actually visits. Given these evaluation points, I then search for the network weights θ that minimize the sum of squared residuals.

   Recall that I draw my evaluation points stochastically because using a grid is infeasible in high dimensions. In a one-dimensional model like this application, a grid would suffice; however, I use stochastic sampling here to introduce an approach that becomes essential in higher-dimensional applications.

8) For optimization, I employ ADAM using all 100 evaluation points (no need, in this simple problem, to use a minibatch). ADAM is a reliable default optimizer for this class of problems.

9) I use 100.000 epochs to ensure the loss function converges to a small value (approximately 10^{-5}). A common practice is to begin with a large number of epochs and check convergence by slightly increasing it. If the loss does not improve meaningfully and is already below a small tolerance, the chosen number of epochs is sufficient.

10) Rather than relying on a held-out validation set, I assess accuracy by comparing against a value function computed using a standard finite difference approach with a very fine grid (and, thus, an excellent approximation to the exact but unknown solution).

11) The code is written in `Python` using `PyTorch`, an open-source and widely used library for machine learning. The implementation runs in about one minute on a standard laptop.[36] The code is self-contained, and in my graduate teaching, I explain it line by line in less than 30 minutes.

Results

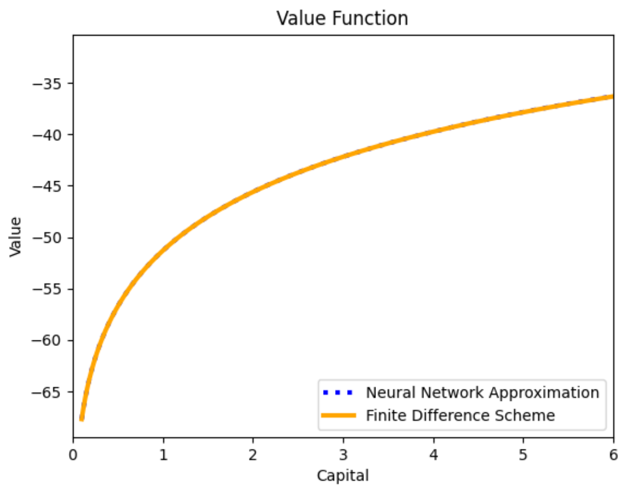

Figure 10. : Value function of the social planner: Neural network approximation and finite difference approximation

Figure 10 plots the approximated value function of the social planner. There are two lines in the graph: the dashed line is the value function from the deep neural network.

[36]The `JAX` library is more flexible and often faster, but it requires familiarity with functional programming, a skill less common among economists. `PyTorch` follows a more conventional object-oriented paradigm and is easier to learn for most users.

The solid line represents the value function computed using a finite-difference method with a very fine grid. The inability to visually separate the lines indicates that the deep neural network learns the value function with high accuracy.

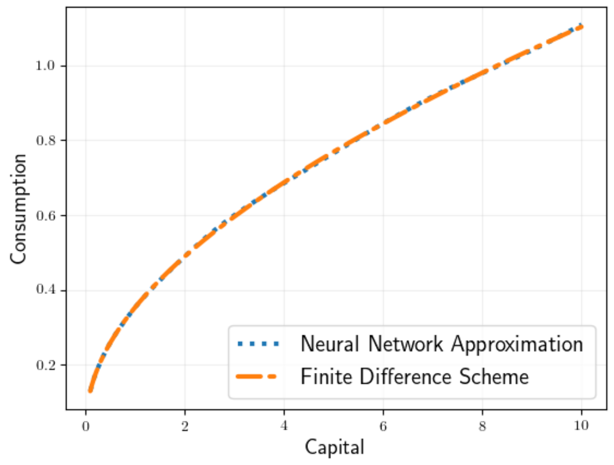

Figure 11. : Decision rule of the social planner for consumption: Neural network approximation and finite difference approximation

Figure 11 plots the approximated decision rule of the social planner for consumption. Again, for comparison, the figure also plots the approximated decision rule obtained with a standard finite-difference approach on a very fine grid.

Figure 12 plots the evolution of the mean squared HJB residual $(\mathrm{err}_{HJB,\theta}(k))^2$ as training progresses.[37] The error remains high for many iterations before decreasing. This is not unusual: the optimizer typically requires some time before moving to the region of the parameter space where learning begins to occur efficiently. A similar figure can be plotted for the decision rule residual $(\mathrm{err}_{c,\theta}(k))^2$.

Possible pitfalls

What are the potential drawbacks of using deep learning to solve dynamic economic models? First, as noted above, we generally lack guarantees of global correctness: there is no assurance that the learned solution corresponds to the model's true global equilibrium.

In practice, the training procedure may converge to a local minimum of the loss function. Moreover, when the equilibrium conditions are ill conditioned, even a small residual in an operator equation such as $\mathcal{T}(d) = 0$ may correspond to an approximate

[37]To make the figure easier to read, I plot a moving average over the last ten iterations, which smooths out sudden spikes.

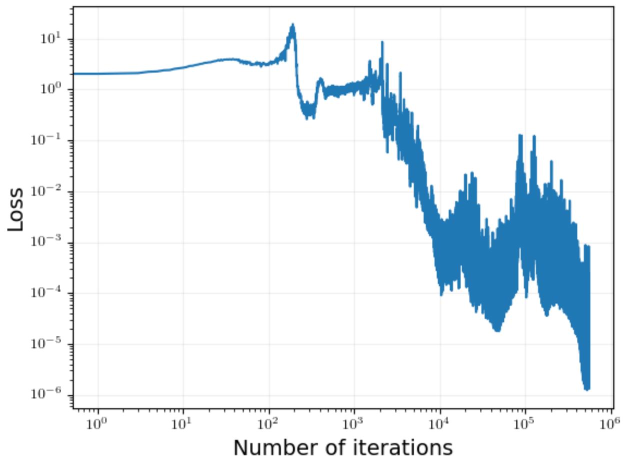

Figure 12. : Evolution of the HJB residual

policy function $\widehat{d}$ that is far from the exact solution d.[38] These limitations have motivated recent work on constrained training schemes and hybrid approaches that combine deep learning with traditional solution algorithms.

Second, training can be unstable and sensitive to hyperparameters, including the learning rate, architecture, initialization, and activation function. Small changes in these choices can alter the results, and convergence may fail or produce poor approximations. Here, adaptive optimization algorithms (e.g., ADAM) and advances in architecture design provide partial remedies, but tuning remains a demanding task.

Third, we lack some of the diagnostic tools available for traditional methods. For instance, in value function iteration, we have well-understood error bounds, whereas in deep learning, error guarantees are looser and often asymptotic, making it harder to evaluate solution quality.

Finally, in many cases, alternative methods, such as perturbation, are easier to implement and can achieve sufficient accuracy at minimal computational cost. Deep learning is not a universal replacement for classical solution methods but rather a complementary tool that extends the set of models we can solve, particularly when dimensionality or nonlinearity renders traditional approaches intractable.

Other applications in economics

The solution of dynamic programming problems is only one of many applications of deep learning to the computation of economic models. Other examples include the computation of models with heterogeneous agents. For example, Fernández-Villaverde et al. (2023) show how to compute the law of motion that describes the evolution of the distribution of assets using neural networks in continuous time,

[38]This concern also applies to projection methods; by contrast, value function iteration offers stronger theoretical guarantees under standard assumptions.

while Fernández-Villaverde et al. (2024) apply the same approach to discrete time. Ebrahimi Kahou et al. (2021), Han et al. (2022), Gu et al. (2023), Payne et al. (2024), Gopalakrishna et al. (2024) follow the proposal in Fernández-Villaverde et al. (2023) in many different applications. Kase et al. (2024) extend the idea to the estimation of dynamic economic models.

There are also applications in computer science to solving models in game theory (e.g., Bichler et al., 2021 and Marris et al., 2022) and in mechanism design (Dütting et al., 2024).

Finally, there are also many applications of reinforcement learning (which I did not have space to cover). Just a few notable ones include Hinterlang and Tänzer (2021), Atashbar and Shi (2022), Chen et al. (2023), and Covarrubias (2023).

10. *Conclusion*

Deep learning is revolutionizing quantitative economics by providing tools for solving high-dimensional, dynamic economic models. By combining function approximation with innovative architectures, deep learning opens many unexplored paths. What we are witnessing today is only the beginning of this process: the work I have reviewed in this survey should be seen as the appetizer for a much larger feast of methods and results yet to come.

However, many open questions remain in the field, particularly those that standard solution methods cannot handle satisfactorily. First, how can we use deep learning to approximate equilibrium correspondences, such as those that appear in dynamic and repeated game theory (Abreu et al., 1990)? Second, how can we employ it to solve for all the equilibria in models with multiple equilibria? Third, how can we use it to address the problem of "forecasting the forecasts of others" (Townsend, 1983)? Fourth, how can deep learning be applied to models with fat tails and rare disasters? Fifth, how can deep learning be applied to solve dynamic stochastic spatial models (Redding, 2024), which are gaining much popularity but remain very hard to compute? Sixth, how can it help address mechanism-design problems involving high-dimensional types? Seventh, can we derive better theoretical bounds on the accuracy of the approximations? Finally, we need to examine the range of architectures in greater detail and assess their relative strengths and weaknesses, particularly in fields where deep learning has not yet been applied as extensively as one might hope, such as industrial organization or international trade.

REFERENCES

Abreu, D., D. Pearce, and E. Stacchetti (1990): "Toward a theory of discounted repeated games with imperfect monitoring," *Econometrica*, 58, 1041–1063.

Aggarwal, C. C. (2020): *Linear Algebra and Optimization for Machine Learning: A Textbook*, Springer, 1st ed.

——— (2023): *Neural Networks and Deep Learning: A Textbook*, Springer, 2nd ed.

Arlot, S. and A. Celisse (2010): "A survey of cross-validation procedures for model selection," *Statistics Surveys*, 4, 40–79.

Atashbar, T. and R. A. Shi (2022): "Deep reinforcement learning: Emerging trends in macroeconomics and future prospects," Working Paper 2022/259, IMF.

Athey, S. and G. W. Imbens (2019): "Machine learning methods that economists should know about," *Annual Review of Economics*, 11, 685–725.

Auclert, A., B. Bardóczy, M. Rognlie, and L. Straub (2021): "Using the sequence-space Jacobian to solve and estimate heterogeneous-agent models," *Econometrica*, 89, 2375–2408.

Azinovic, M., L. Gaegauf, and S. Scheidegger (2022): "Deep equilibrium nets," *International Economic Review*, 63, 1471–1525.

Azinovic, M. and J. Zemlicka (2024): "Intergenerational consequences of rare disasters," Manuscript.

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

Bellman, R. (1957): *Dynamic Programming*, Princeton University Press.

Bichler, M., M. Fichtl, S. Heidekrüger, N. Kohring, and P. Sutterer (2021): "Learning Equilibria in Symmetric Auction Games Using Artificial Neural Networks," *Nature Machine Intelligence*, 3, 687–695.

Brumm, J. and S. Scheidegger (2017): "Using adaptive sparse grids to solve high-dimensional dynamic models," *Econometrica*, 85, 1575–1612.

Candler, G. V. (2001): "Finite-difference methods for continuous-time dynamic programming," in *Computational Methods for the Study of Dynamic Economies*, ed. by R. Marimón and A. Scott, Oxford University Press.

Chakraborty, C. and A. Joseph (2017): "Machine learning at central banks," Working Paper 674, Bank of England.

Chen, M., A. Joseph, M. Kumhof, X. Pan, and X. Zhou (2023): "Deep reinforcement learning in a monetary model," Tech. Rep. 2104.09368, arXiv.

Covarrubias, M. (2023): "Dynamic oligopoly and monetary policy: A deep reinforcement learning approach," Manuscript.

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

David, P. A. (1985): "Clio and the economics of QWERTY," *American Economic Review*, 75, 332–337.

de Araujo, D. K. G., S. Doerr, L. Gambacorta, and B. Tissot (2024): "Artificial intelligence in central banking," BIS Bulletins 84, Bank for International Settlements.

Dell, M. (2025): "Deep learning for economists," *Journal of Economic Literature*, 63, 5–58.

DEN HAAN, W. J. AND A. MARCET (1990): "Solving the Stochastic Growth Model by Parameterizing Expectations," *Journal of Business & Economic Statistics*, 8, 31–34.

DU, S., J. LEE, H. LI, L. WANG, AND X. ZHAI (2019): "Gradient Descent Finds Global Minima of Deep Neural Networks," in *Proceedings of the 36th International Conference on Machine Learning*, ed. by K. Chaudhuri and R. Salakhutdinov, vol. 97 of *Proceedings of Machine Learning Research*, 1675–1685.

DUARTE, V., D. DUARTE, AND D. SILVA (2024): "Machine learning for continuous-time finance," *Review of Financial Studies*, 11, 3217–3271.

DÜTTING, P., Z. FENG, H. NARASIMHAN, D. C. PARKES, AND S. S. RAVINDRANATH (2024): "Optimal Auctions through Deep Learning: Advances in Differentiable Economics," *Journal of the ACM*, 71, 5:1–5:53.

EBRAHIMI KAHOU, M., J. FERNÁNDEZ-VILLAVERDE, S. GOMEZ-CARDONA, J. PERLA, AND J. ROSA (2024): "Spooky boundaries at a distance: Inductive bias, dynamic models, and behavioral macro," Working Paper 32850, National Bureau of Economic Research.

EBRAHIMI KAHOU, M., J. FERNÁNDEZ-VILLAVERDE, J. PERLA, AND A. SOOD (2021): "Exploiting symmetry in high-dimensional dynamic programming," Working Paper 28981, National Bureau of Economic Research.

FERNÁNDEZ-VILLAVERDE, J., S. HURTADO, AND G. NUÑO (2023): "Financial frictions and the wealth distribution," *Econometrica*, 91, 869–901.

FERNÁNDEZ-VILLAVERDE, J., J. MARBET, G. NUÑO, AND O. RACHEDI (2024): "Inequality and the zero lower bound," *Journal of Econometrics*, 105819.

FERNÁNDEZ-VILLAVERDE, J., G. NUÑO, G. SORG-LANGHANS, AND M. VOGLER (2020): "Solving high-dimensional dynamic programming problems using deep learning," Manuscript.

FERNÁNDEZ-VILLAVERDE, J., J. F. RUBIO-RAMÍREZ, AND F. SCHORFHEIDE (2016): "Solution and estimation methods for DSGE models," in *Handbook of Macroeconomics*, Elsevier, vol. 2, 527–724.

FOLINI, D., A. FRIEDL, F. KÜBLER, AND S. SCHEIDEGGER (2024): "The climate in climate economics," *Review of Economic Studies*, rdae011.

FUJIMOTO, S., D. MEGER, D. PRECUP, O. NACHUM, AND S. S. GU (2022): "Why should I trust you, Bellman? The Bellman error is a poor replacement for value error," in *Proceedings of the 39th International Conference on Machine Learning*, vol. 162, 6918–6943.

GOPALAKRISHNA, G. (2021): "ALIENs and continuous time economies," Swiss Finance Institute Research Paper Series 21-34, Swiss Finance Institute.

GOPALAKRISHNA, G., Z. GU, AND J. PAYNE (2024): "Institutional asset pricing, segmentation, and inequality," Manuscript.

Gorodnichenko, Y., S. Maliar, and C. Naubert (2020): "Household savings and monetary policy under individual and aggregate stochastic volatility," Discussion paper 15614, CEPR.

Goujon, A., A. Etemadi, and M. Unser (2024): "On the number of regions of piecewise linear neural networks," *Journal of Computational and Applied Mathematics*, 441, 115667.

Gu, Z., M. Lauriere, S. Merkel, and J. Payne (2023): "Global solutions to master equations for continuous time heterogeneous agent macroeconomic models," SSRN 4871228.

Han, J., Y. Yang, and W. E (2022): "DeepHAM: A global solution method for heterogeneous agent models with aggregate shocks," Tech. Rep. 2112.14377, arXiv.

Hastie, T., R. Tibshirani, and J. Friedman (2009): *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, New York: Springer, 2 ed.

Hinterlang, N. and A. Tänzer (2021): "Optimal monetary policy using reinforcement learning," Discussion Papers 51/2021, Deutsche Bundesbank.

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

Huang, J. (2023): "Breaking the curse of dimensionality in heterogeneous-agent models: A deep learning-based probabilistic approach," Available at SSRN 4649043.

Jacot, A., S. H. Choi, and Y. Wen (2024): "How DNNs break the curse of dimensionality: Compositionality and symmetry learning," Tech. rep., arXiv.

Judd, K. L. (1998): *Numerical Methods in Economics*, MIT Press.

Kase, H., L. Melosi, and M. Rottner (2024): "Estimating nonlinear heterogeneous agent models with neural networks," Research Paper Series 1499, University of Warwick, Department of Economics.

Kelly, B. T. and D. Xiu (2023): "Financial machine learning," NBER Working Papers 31502, National Bureau of Economic Research.

Kendall, A., Y. Gal, and R. Cipolla (2018): "Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 7482–7491.

Kidger, P. and T. Lyons (2020): "Universal approximation with deep narrow networks," Tech. rep., arXiv.

Kingma, D. P. and J. Ba (2017): "Adam: A method for stochastic optimization," Tech. Rep. 1412.6980, arXiv.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012): "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 1097–1105.

Maliar, L., S. Maliar, and P. Winant (2021): "Deep learning for solving dynamic economic models," *Journal of Monetary Economics*, 122, 76–101.

Marris, L., I. Gemp, T. Anthony, A. Tacchetti, S. Liu, and K. Tuyls (2022): "Turbocharging Solution Concepts: Solving NEs, CEs and CCEs with Neural Equilibrium Solvers," in *Advances in Neural Information Processing Systems*, vol. 35, 5586–5600.

McCulloch, W. S. and W. Pitts (1943): "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, 5, 115–133.

Mei, S., A. Montanari, and P.-M. Nguyen (2018): "A mean field view of the landscape of two-layer neural networks," *Proceedings of the National Academy of Sciences*, 115, E7665–E7671.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis (2015): "Human-level control through deep reinforcement learning," *Nature*, 518, 529–533.

Murphy, K. P. (2022): *Probabilistic Machine Learning: An Introduction*, MIT Press.

——— (2024): *Probabilistic Machine Learning: Advanced Topics*, MIT Press.

Nagel, S. (2021): *Machine Learning in Asset Pricing*, Princeton University Press.

Newell, A., H. A. Simon, and C. Shaw (1956): "Logic Theorist," Computer program.

Nuño, S. Scheidegger, and P. Renner (2024): "Let bygones be bygones: Optimal monetary policy with persistent supply shocks," Manuscript.

Payne, J., A. Rebei, and Y. Yang (2024): "Deep learning for search and matching models," Manuscript.

Poggio, T., H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao (2017): "Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review," *International Journal of Automation and Computing*, 14, 503–519.

Prince, S. J. (2023): *Understanding Deep Learning*, MIT Press.

Redding, S. J. (2024): "Spatial Economics," Working Paper 33125, National Bureau of Economic Research.

Robbins, H. and S. Monro (1951): "A stochastic approximation method," *Annals of Mathematical Statistics*, 22, 400–407.

Rosenblatt, F. (1958): "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, 65, 386–408.

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

Russell, S. and P. Norvig (2020): *Artificial Intelligence: A Modern Approach*, Pearson, 4th ed.

Sauzet, M. (2021): "Projection methods via neural networks for continuous-time models," Manuscript.

Scheidegger, S. and I. Bilionis (2019): "Machine learning for high-dimensional dynamic stochastic economies," *Journal of Computational Science*, 33, 68–82.

Soudry, D., E. Hoffer, M. S. Nacson, S. Gunasekar, and N. Srebro (2018): "The Implicit Bias of Gradient Descent on Separable Data," *Journal of Machine Learning Research*, 19, 1–57.

Stokey, N. L., R. E. Lucas, and E. C. Prescott (1989): *Recursive Methods in Economic Dynamics*, Cambridge, MA: Harvard University Press.

Sutton, R. S. and A. G. Barto (2018): *Reinforcement Learning*, MIT Press, 2nd ed.

Townsend, R. M. (1983): "Forecasting the forecasts of others," *Journal of Political Economy*, 91, 546–588.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017): "Attention is all you need," in *Advances in Neural Information Processing Systems*, 5998–6008.

Villa, A. T. and V. Valaitis (2024): "Machine learning projection method for macro-finance models," *Quantitative Economics*, 15, 145–173.

Wilson, A. G. (2025): "Deep learning is not so mysterious or different," in *Forty-second International Conference on Machine Learning Position Paper Track*.

Wooldridge, M. (2021): *A Brief History of Artificial Intelligence: What It Is, Where We Are, and Where We Are Going*, Flatiron Books.

Zhang, A., Z. C. Lipton, M. Li, and A. J. Smola (2023): *Dive into Deep Learning*, Cambridge University Press.

Zheng, A. and A. Casari (2018): *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*, O'Reilly Media.