Solving the neoclassical growth model with a neural network

Gustavo Mellior*

December 2021

* University of Liverpool. Email address: G.Mellior@liverpool.ac.uk

The problem

The problem is to solve

$$\max_{c_t} \int_0^\infty u(c_t)\mathrm{d}t$$
$$\text{subject to} \quad \dot{k_t} = Ak_t^\alpha - \delta k_t - c_t. \tag{1}$$

The solution to this problem satisfies the Hamilton-Jacobi-Bellman (HJB) equation, shown below in (2). The value function V is a function of time t and capital $k(t)$. I omit explicit dependence on these variables in the notation unless it is absolutely necessary to show this, i.e. $V(k(t), t) = V$. Let's represent the partial derivative of V with respect to k as V_k. V_t is the partial derivative of V with respect to time. The HJB equation is

$$\rho V = \max_c + u(c) + V_k \left[Ak^\alpha - \delta k - c\right] + V_t, \tag{2}$$

where ρ, α, δ and A are positive constants. Consumption is represented by c. Let $u(c) = \dfrac{c^{1-\sigma}}{1-\sigma}$. The first order condition (F.O.C.) is given by

$$c = V_k^{-\frac{1}{\sigma}}. \tag{3}$$

The F.O.C. tells us how much to consume and as a by-product how the capital stock evolves theough time. As we are solving for the stationary HJB equation we can set $V_t = 0$. Hence, we are solving for the following ODE in terms of k.

$$\rho V = \frac{\sigma}{1-\sigma} V_k^{\frac{\sigma-1}{\sigma}} + V_k \left[Ak^\alpha - \delta k\right] \tag{4}$$

Boundary condition

We need to define the boundary condition. We can pick this at $k = k^*$, the steady state capital stock, as it is intuitive to find what consumption c is at that point in the state space. By doing so we can pin down what V_k is at k^*. In steady state we have that $\dot{k} = Ak^{*^\alpha} - \delta k^* - c(k^*) = 0$. Hence, $c(k^*) = Ak^{*^\alpha} - \delta k^*$. So our boundary condition is given by the next expression.

$$V_k(k^*) = \left(Ak^{*^\alpha} - \delta k^*\right)^{-\sigma} \tag{5}$$

Translating the HJB equation into Julia

We can solve for V and c with two separate approaches. One approach creates two separate neural networks (one for V and one for c). We update the coefficients of the neural networks so as to minimize the HJB equation errors and the F.O.C. errors $c - V_k^{-\frac{1}{\sigma}}$, respectively[1]. In this example we approximate V with a neural network directly. Then, we differentiate the resulting neural network of V with respect to its input, in this case, capital, so as to obtain c. Both approaches yield identical results although I find the latter to deliver the result in substantially less iterations (often in less than 700 instead of roughly 3600 iterations).

We solve for V with the boundary condition mentioned above. In order for the solution to make sense[2] we need to make sure that $c > 0$. In other words, we need to make sure that the slope of the value function is always positive. We can do that by using the max function as shown below. Note that ϵ is a small positive number.

$$\frac{\sigma}{1-\sigma} \max\{V_k, \epsilon\}^{\frac{\sigma-1}{\sigma}} + \max\{V_k, \epsilon\} \left[Ak^\alpha - \delta k\right] - \rho V = \text{HJB error} \simeq 0 \tag{6}$$

If we want to solve for V and c by using a neural network approximation for each function, then we minimize the errors of the following two expressions.

[1] The closest reference to this example is Fernández-Villaverde et al. (2020). Another reference using different methods but relying on deep learning to solve high dimensional HJB equations is Duarte (2018).

[2] Negative consumption makes no sense. Not preventing this will return results with complex numbers.

$$\frac{\sigma}{1-\sigma}\max\{c,\epsilon\}^{1-\sigma}+\max\{V_k,\epsilon\}\left[Ak^{\alpha}-\delta k\right]-\rho V=\text{HJB error}\simeq 0$$
$$\max\{V_k^{-\frac{1}{\sigma}},\epsilon\}-c=\text{FOC error}\simeq 0 \tag{7}$$

And as before we use the boundary condition defined in (5).

Setting up the neural network

The problem is solved in Julia, using its NeuralPDE solvers Zubov et al. (2021). The code `SolveHJBandpolicy.jl` illustrates how to set up the problem, the neural network and pass it on to the solver[3]. I find that using a neural network with three hidden layers, where each layer has 20 neurons, and where the first two hidden layers use a tanh activation function followed by a softplus yield the best results. I use the ADAM optimizer with a learning rate of 0.06 until the loss dips below 1e-7 and then do a few more hundred iterations with BFGS, until the loss dips below 1e-10. Julia's packages are already optimized to initialize parameter weights for the neural net. For instance, Xavier initialization is set as the default when we build a neural network with Julia's `FastChain` command.

Calibration

$\rho = 0.05$, $\delta = 0.05$, $\alpha = 0.34$, $A = 1$, $\epsilon = 0.001$ and $\sigma = 2$.

Results

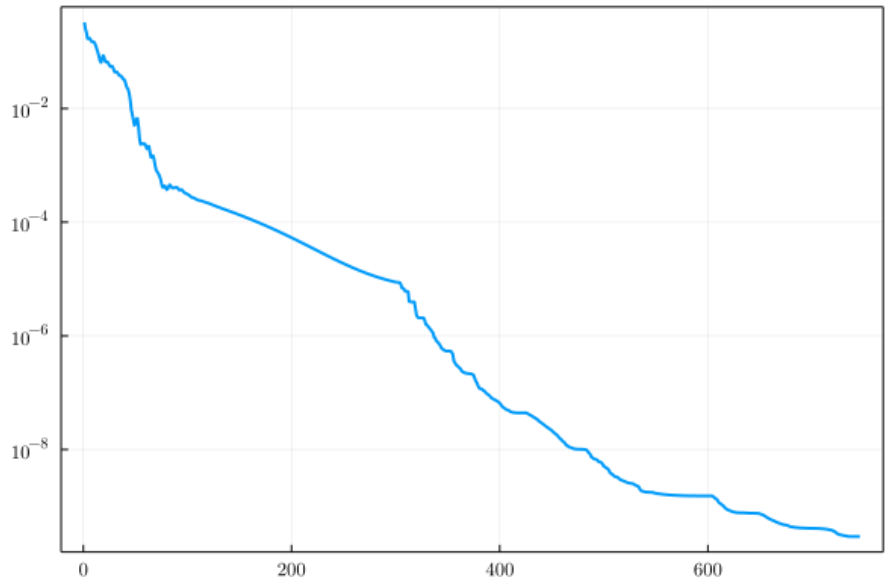

Figure 1: Evolution of the loss

[3]Sometimes the neural network does not deliver the correct solution. If this happens simply try running the code again.

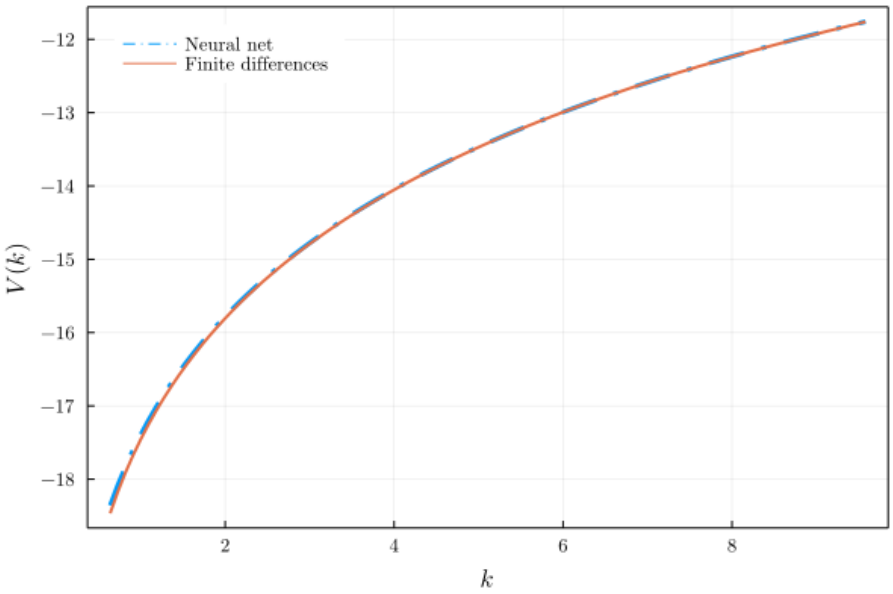

Figure 2: Value function

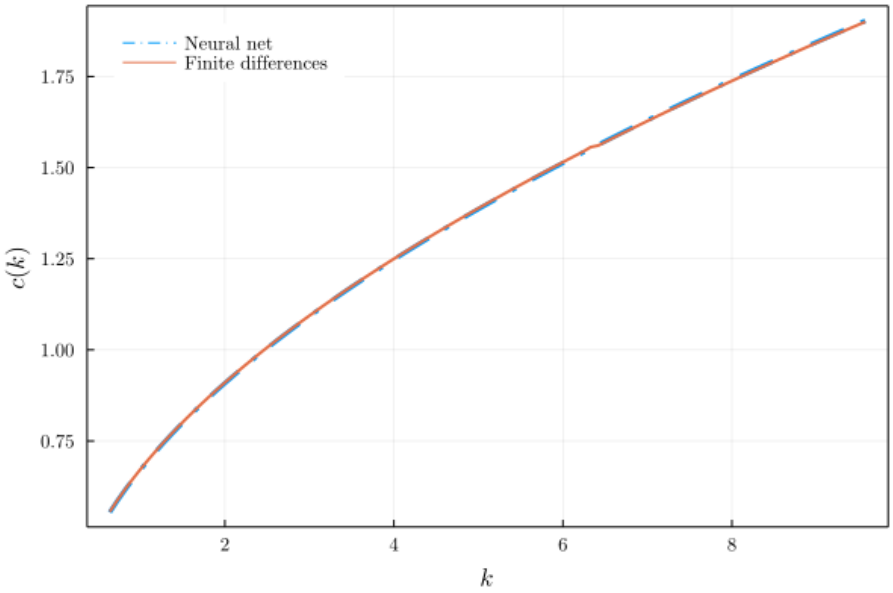

Figure 3: Consumption

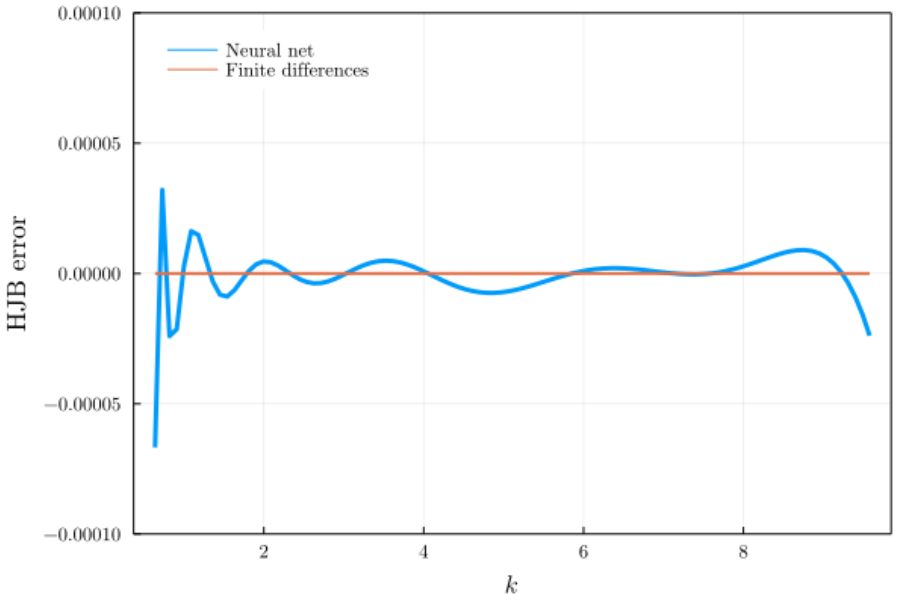

Figure 4: HJB equation error

References

Duarte, V. (2018): "Machine learning for continuous-time economics," *Working paper*.

Fernández-Villaverde, J., G. Nuño, G. Sorg-Langhans, and M. Vogler (2020): "Solving high-dimensional dynamic programming problems using deep learning," *Working paper*.

Zubov, K., Z. McCarthy, Y. Ma, F. Calisto, V. Pagliarino, S. Azeglio, L. Bottero, E. Luján, V. Sulzer, A. Bharambe, et al. (2021): "NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations," *arXiv preprint arXiv:2107.09443*.