Machine Learning for Macrofinance

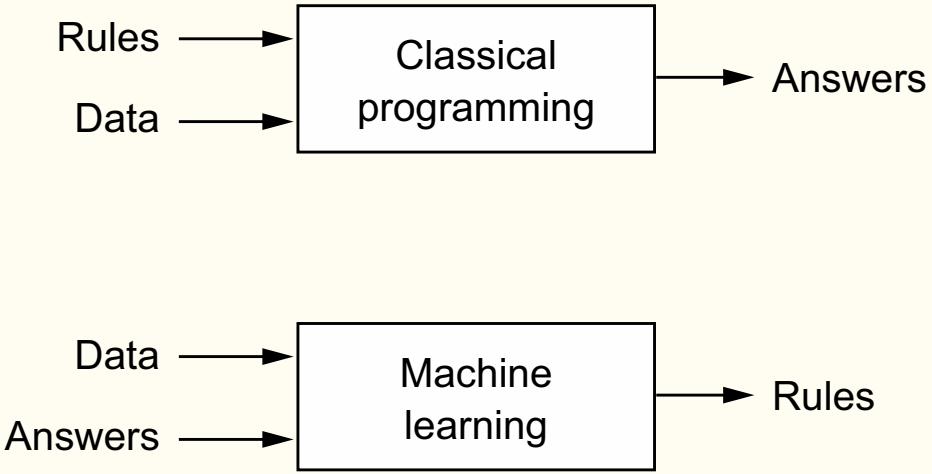
Jesús Fernández-Villaverde[1]

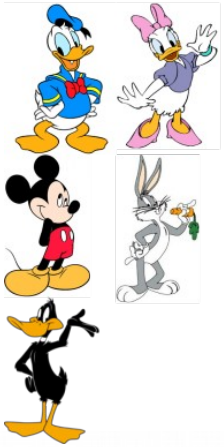
August 8, 2022

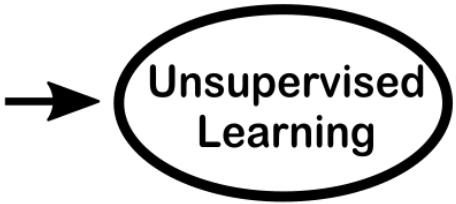
[1]University of Pennsylvania

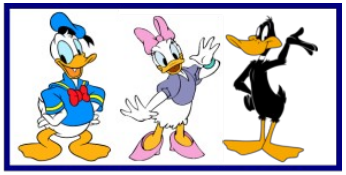
What is machine learning?

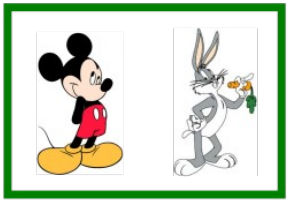
- Wide set of algorithms to detect and learn from patterns in the data (observed or simulated) and use them for decision making or to forecast future realizations of random variables.
- Focus on recursive processing of information to improve performance over time.
- In fact, this is clearer to see in its name in other languages: Apprentissage automatique or aprendizaje automático.
- Even in English: Statistical learning.
- More formally: we use rich datasets to select appropriate functions in a dense functional space.

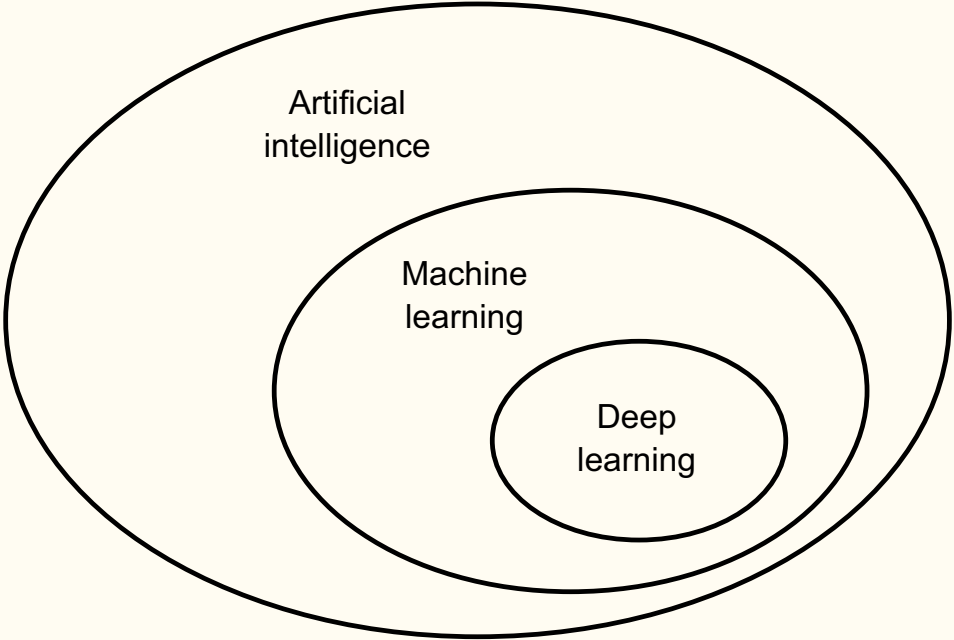

Rules
Data
Classical programming
Answers
Data
Answers
Machine learning
Rules




Unsupervised
Learning

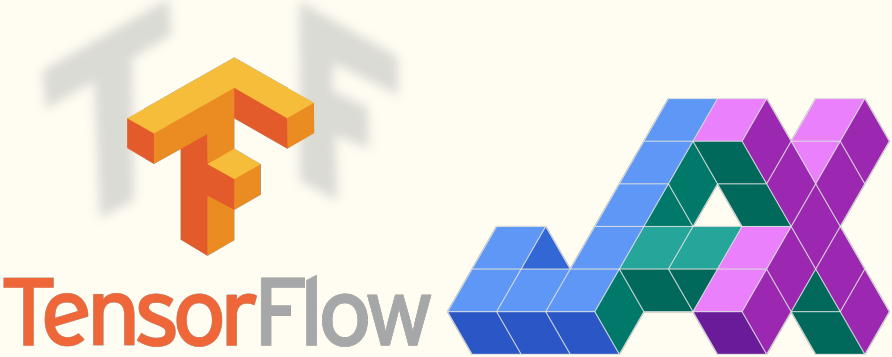





Artificial
intelligence
Machine
learning
Deep
learning

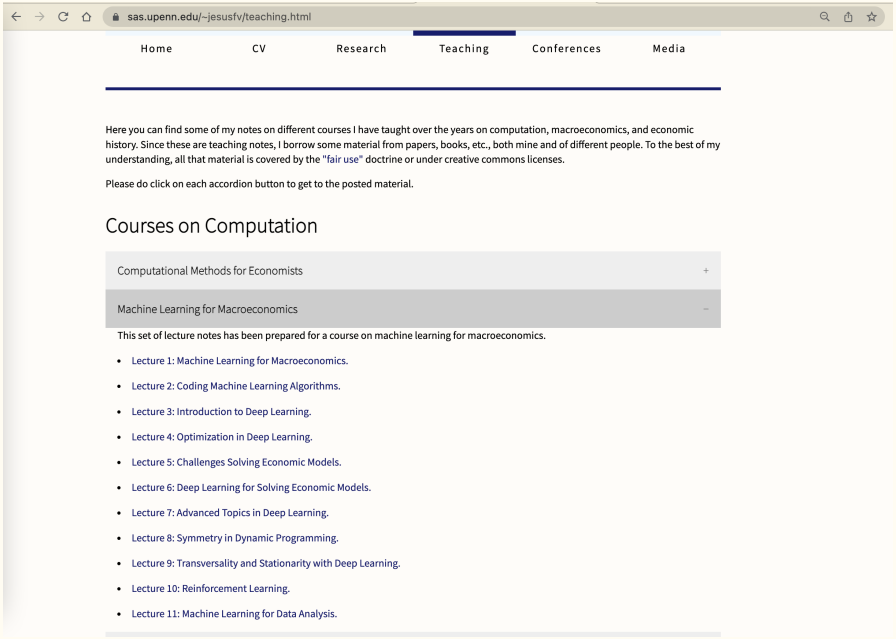
Why now?

- Many of the ideas of machine learning (e.g., basic neural network by McCulloch and Pitts, 1943, and perceptron by Rosenblatt, 1958) are decades old.
- Previous waves of excitement followed by backlashes.
- Four forces behind the revival:
  1. Big data.
  2. Long tails.
  3. Cheap computational power.
  4. Algorithmic advances.
- Likely that these four forces will become stronger over time.
- Exponential growth in industry⇒plenty of libraries for `Python`, `R`, and other languages.


TensorFlow
PyTorch
flux

The many uses of machine learning in macrofinance

- Recent boom in economics:
  1. New solution methods for economic models: my own work on deep learning.
  2. Alternative to older bounded rationality models: reinforcement learning.
  3. Data processing: Blumenstock *et al.* (2017).
  4. Alternative empirical models: deep IVs by Hartford *et al.* (2017) and text analysis.
- However, important to distinguish signal from noise.
- Machine learning is a catch-all name for a large family of methods.
- Some of them are old-fashioned methods in statistics and econometrics presented under alternative names.


sas.upenn.edu/~jesusfv/teaching.html
Home
CV
Research
Teaching
Conferences
Media
Here you can find some of my notes on different courses I have taught over the years on computation, macroeconomics, and economic history. Since these are teaching notes, I borrow some material from papers, books, etc., both mine and of different people. To the best of my understanding, all that material is covered by the "fair use" doctrine or under creative commons licenses.
Please do click on each accordion button to get to the posted material.
Courses on Computation
Computational Methods for Economists
Machine Learning for Macroeconomics
This set of lecture notes has been prepared for a course on machine learning for macroeconomics.
Lecture 1: Machine Learning for Macroeconomics.
Lecture 2: Coding Machine Learning Algorithms.
Lecture 3: Introduction to Deep Learning.
Lecture 4: Optimization in Deep Learning.
Lecture 5: Challenges Solving Economic Models.
Lecture 6: Deep Learning for Solving Economic Models.
Lecture 7: Advanced Topics in Deep Learning.
Lecture 8: Symmetry in Dynamic Programming.
Lecture 9: Transversality and Stationarity with Deep Learning.
Lecture 10: Reinforcement Learning.
Lecture 11: Machine Learning for Data Analysis.

A formal approach

The problem

- Let us suppose we want to approximate ("learn") an unknown function:

$$y = f(\mathbf{x})$$

  where y is a scalar and $\mathbf{x} = \{x_0 = 1, x_1, x_2, ..., x_N\}$ a vector (why a constant?).

- We care about the case when N is large (possibly in the thousands!).

- Easy to extend to the case where y is a vector (e.g., a probability distribution), but notation becomes cumbersome.

- In economics, $f(\mathbf{x})$ can be a value function, a policy function, a pricing kernel, a conditional expectation, a classifier, ...

A neural network

- A neural network is an approximation to $f(\mathbf{x})$ of the form:

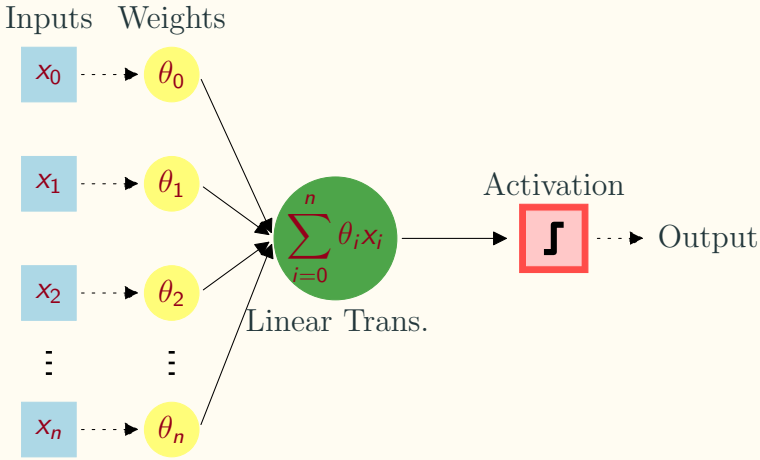
$$y \cong g^{NN}(\mathbf{x}; \theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi(z_m)$$

  where $\phi(\cdot)$ is an activation function and:

$$z_m = \sum_{n=0}^{N} \theta_{n,m} x_n$$

- The x_n's are known as the features of the data, which belong to a feature space $\mathcal{X}$.
- The $\phi(z_m)$'s are known as the representation of the data.
- M is known as the width of the model (wide vs. thin networks).
- "Training" the network: selecting θ such that $g^{NN}(\mathbf{x}; \theta)$ is as close to $f(\mathbf{x})$ as possible given some relevant metric (e.g., the ℓ_2 norm).

Flow representation



Comparison with other approximations

- Compare:

$$y \cong g^{NN}(\mathbf{x};\theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi \left(\sum_{n=0}^{N} \theta_{n,m} x_n \right)$$

  with a standard projection:

$$y \cong g^{CP}(\mathbf{x};\theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(\mathbf{x})$$

  where ϕ_m is, for example, a Chebyshev polynomial.

- We exchange the rich parameterization of coefficients for the parsimony of basis functions.

- In a few slides, I will explain why this is often a good idea. Suffice it to say now that evaluating a neural network is straightforward.

- How we determine the coefficients is also different, but this is less important.

Deep learning I

- A deep learning network is a *multilayer* composition of $J > 1$ neural networks:

$$z_m^0 = \theta_{0,m}^0 + \sum_{n=1}^{N} \theta_{n,m}^0 x_n$$

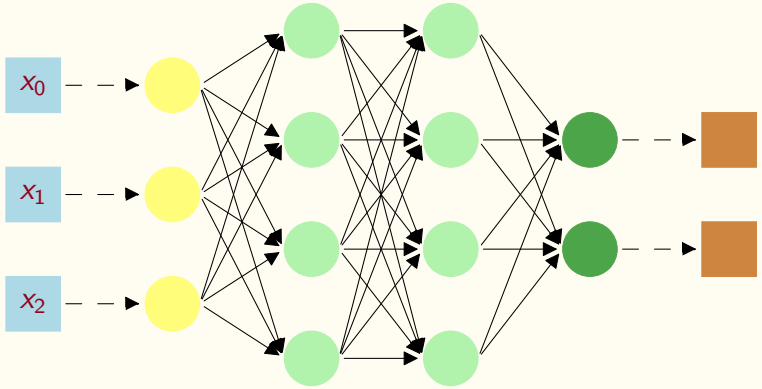
and

$$z_m^1 = \theta_{0,m}^1 + \sum_{m=1}^{M^{(1)}} \theta_m^1 \phi^1 \left(z_m^0 \right)$$

$$\ldots$$

$$y \cong g^{DL}(\mathbf{x}; \theta) = \theta_0^J + \sum_{m=1}^{M^{(J)}} \theta_m^J \phi^J \left(z_m^{J-1} \right)$$

where the $M^{(1)}, M^{(2)}, \ldots$ and $\phi^1(\cdot), \phi^2(\cdot), \ldots$ are possibly different across each layer of the network.



Input Values Hidden Layer 1 Output Layer

Input Layer Hidden Layer 2

Deep learning II

- J is known as the depth of the network (deep vs. shallow networks). The case $J = 1$ is the neural network we saw before.
- From now on, we will refer to neural networks as including both single and multilayer networks.
- As before, we select θ such that $g^{DL}(\mathbf{x}; \theta)$ approximates a target function $f(\mathbf{x})$ as closely as possible under some relevant metric.
- We can also add multidimensional outputs.
- Or even to produce a probability distribution as output, for example, using a softmax layer:

$$y_m = \frac{e^{z_m^{J-1}}}{\sum_{m=1}^{M} e^{z_m^{J-1}}}$$

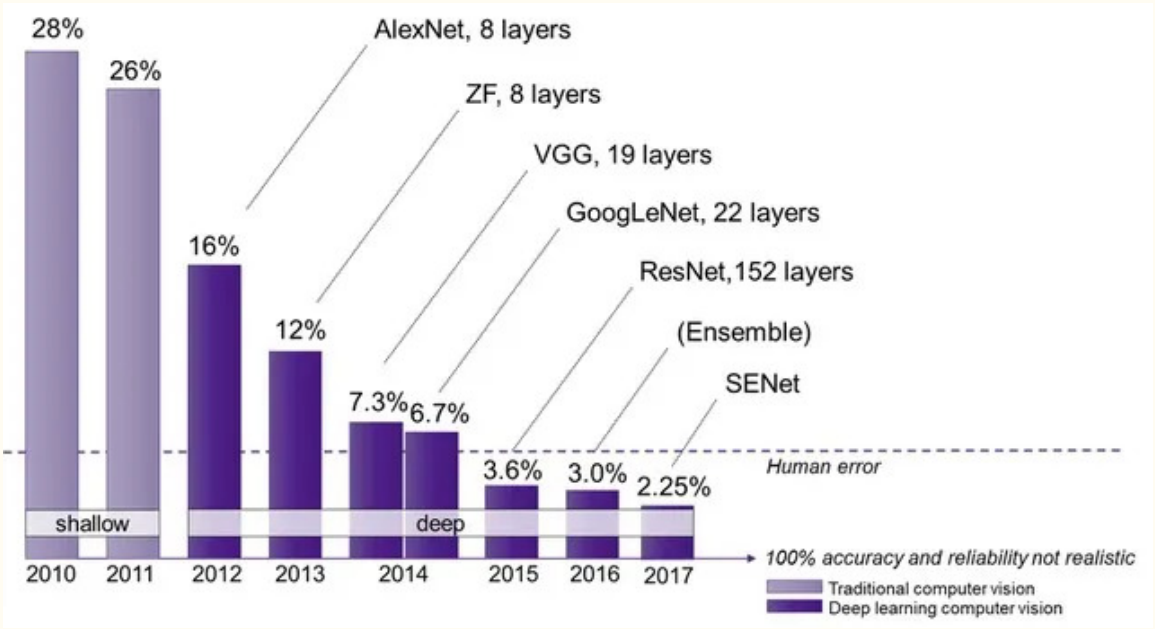
- All other aspects (selecting $\phi(\cdot)$, J, M, ...) are known as the network architecture. We will discuss extensively at the of this slide block how to determine them.

Why do neural networks "work"?

- Neural networks consist entirely of chains of tensor operations: we take $\mathbf{x}$, we perform affine transformations, and apply an activation function.
- Thus, these tensor operations are geometric transformations of $\mathbf{x}$.
- In other words: a neural network is a complex geometric transformation in a high-dimensional space.
- Deep neural networks look for convenient geometrical representations of high-dimensional manifolds.
- The success of any functional approximation problem is to search for the right geometric space in which to perform it, not to search for a "better" basis function.

Why do deep neural networks “work” better?

- Why do we want to introduce hidden layers?
  1. It works! Evolution of ImageNet winners.
  2. The number of representations increases exponentially with the number of hidden layers while computational cost grows linearly.
  3. Intuition: hidden layers induce highly nonlinear behavior in the joint creation of representations without the need to have domain knowledge (used, in other algorithms, in some form of greedy pre-processing).


28%
26%
16%
12%
7.3%
6.7%
3.6%
3.0%
2.25%
AlexNet, 8 layers
ZF, 8 layers
VGG, 19 layers
GoogLeNet, 22 layers
ResNet,152 layers
(Ensemble)
SENet
Human error
shallow
deep
2010
2011
2012
2013
2014
2015
2016
2017
100% accuracy and reliability not realistic
Traditional computer vision
Deep learning computer vision

Some consequences

- Because of the previous arguments, neural networks can efficiently approximate extremely complex functions.
- In particular, under certain (relatively weak) conditions:
  1. Neural networks are universal approximators.
  2. Neural networks break the "curse of dimensionality."
- Furthermore, neural networks are easy to code, stable, and scalable for multiprocessing (neural networks are built around tensors).
- The richness of an ecosystem is key for its long-run success.

The challenges in macrofinance

- Many interesting questions in macrofinance require:
  1. Many state variables. Examples: Corporate finance models, discrete node models, rich life-cycle models, models where parameters are quasi-states.
  2. Nonlinearities (i.e., local features and irregularly-shaped domains). Examples: How do financial crises arise? Why do countries or firms default? When do firms invest in large, lumpy projects?
  3. Heterogeneous agents (i.e., large amounts of data). Examples: What mechanisms account for changes in income and wealth inequality? What is the relation between asset prices and wealth inequality? How does inequality affect monetary and fiscal policy?
- Often, all three elements come together. Example: heterogeneous agents models with binding constraints, nominal frictions, and many assets.

Why are neural networks a good solution method in macrofinance?

Approximation method	Many state variables	Can solve local features accurately	Irregularly shaped domain	Large amount of data
Polynomials	✓	✗	✓	✓
Splines	✗	✓	✗	✓
Adaptive (sparse) grids	✓	✓	✗	✓
Gaussian processes	✓	✓	✓	✗
Deep learning	✓	✓	✓	✓

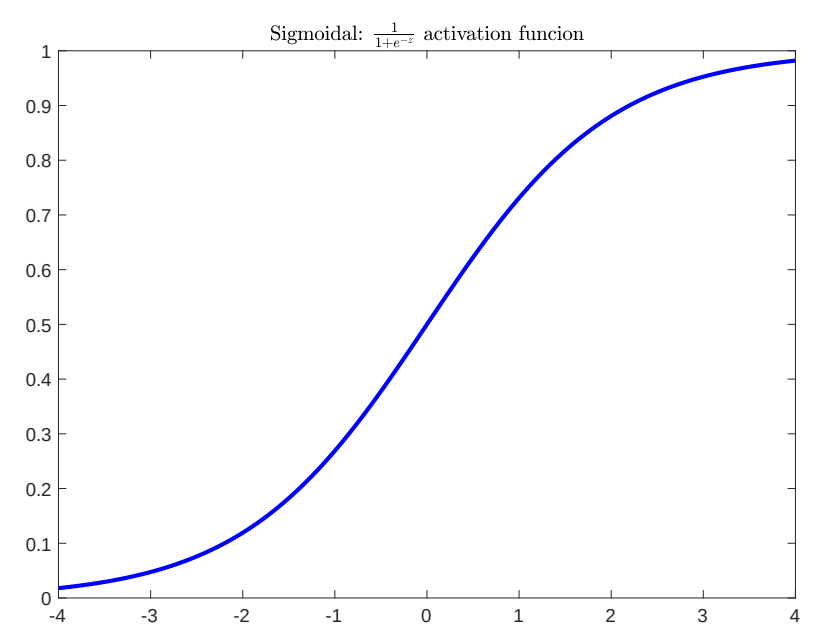
Limitations of neural networks and deep learning

- While deep learning can work extremely well, there is no such a thing as a silver bullet.
- Clear and serious trade-offs in real-life applications.
- We often require tens of thousands of observations to properly train a deep network.
- Of course, sometimes "observations" are endogenous (we can simulate them) and we can implement data augmentation, but if your goal is to forecast GDP next quarter, it is unlikely a deep neural network will beat an ARIMA(n,p,q) (at least only with macro variables).

More details

Activation functions I

- Traditionally:
  1. Identity function:
  $$\phi(z) = z$$
  Used in linear regression.
  2. A sigmoidal function:
  $$\phi(z) = \frac{1}{1 + e^{-z}}$$
  3. Step function (a limiting case as z grows quickly):
  $$\phi(z) = 1 \text{ if } z > 0, \phi(z) = 0 \text{ otherwise.}$$
  4. Hyperbolic tangent:
  $$\phi(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$


Sigmoidal: $\frac{1}{1+e^{-z}}$ activation funcion
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-4
-3
-2
-1
0
1
2
3
4

Activation functions II

- Some activation functions that have gained popularity recently:

1. Rectified linear unit (ReLU):

$$\phi(z) = \max(0, z)$$

2. Parametric ReLU:

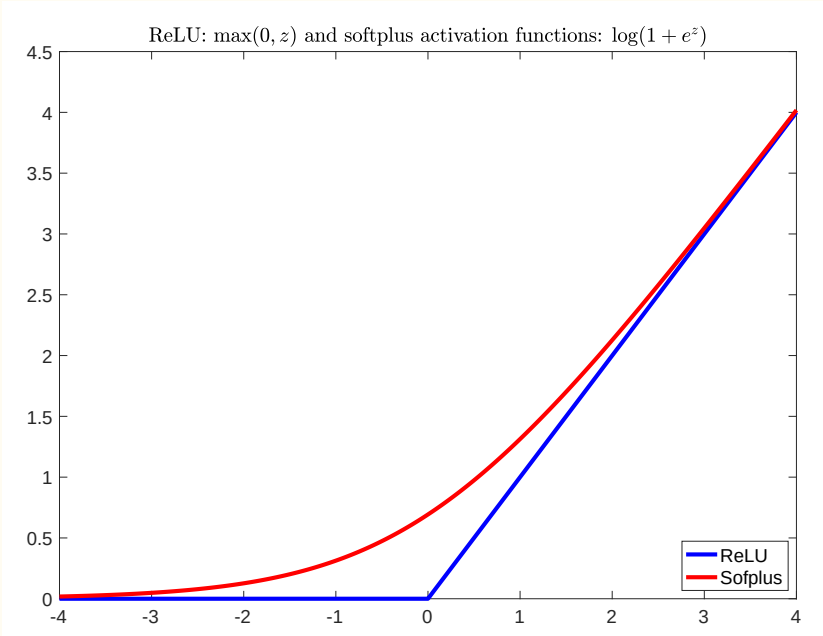
$$\phi(z) = \max(z, az)$$

3. Continuously Differentiable Exponential Linear Units (CELU):

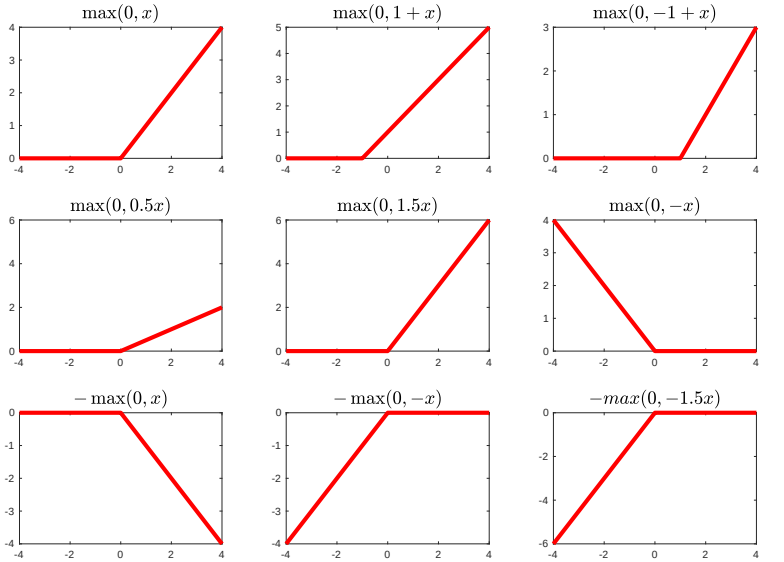
$$\phi(z) = \max(0, z) + \min(0, \alpha(e^{x/\alpha} - 1))$$

4. Softplus:

$$\phi(z) = log(1 + e^{z})$$


ReLU: $\max(0, z)$ and softplus activation functions: $\log(1 + e^z)$
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
-4
-3
-2
-1
0
1
2
3
4
ReLU
Sofplus

Different ReLUs: $\theta_i \max(0, \theta_{i,0} + \theta_{i,1} x)$



Two classic (yet remarkable) results I

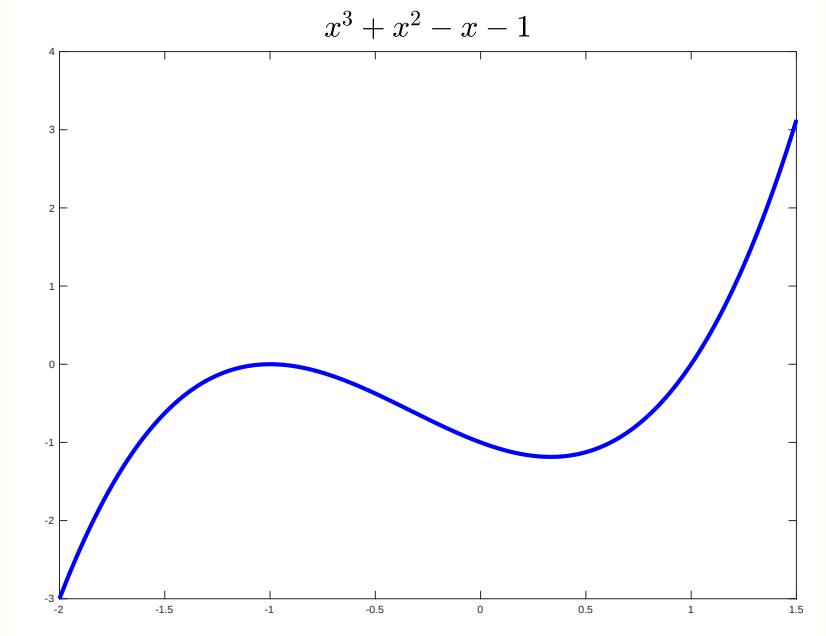
Borel measurable function

A map $f : X \to Y$ between two topological spaces is called Borel measurable if $f^{-1}(A)$ is a Borel set for any open set A on Y (the Borel sets are all the open sets built through the operations of countable union, countable intersection, and relative complement).

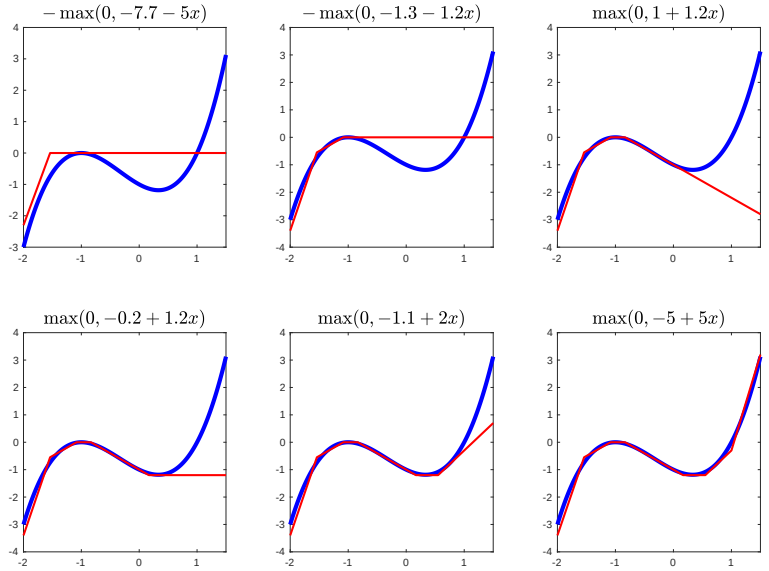
Universal approximation theorem: Hornik, Stinchcombe, and White (1989)

A neural network with at least one hidden layer can approximate any Borel measurable function mapping finite-dimensional spaces to any desired degree of accuracy.

- Intuition of the result.
- Comparison with other results in series approximations.


$x^3 + x^2 - x - 1$
4
3
2
1
0
-1
-2
-3
-2
-1.5
-1
-0.5
0
0.5
1
1.5

A six ReLUs approximation



Two classic (yet remarkable) results II

- Assume, as well, that we are dealing with the class of functions for which the Fourier transform of their gradient is integrable.

Breaking the curse of dimensionality: Barron (1993)

A one-layer NN achieves integrated square errors of order $\mathcal{O}(1/M)$, where M is the number of nodes. In comparison, for series approximations, the integrated square error is of order $\mathcal{O}(1/(M^{2/N}))$ where N is the dimensions of the function to be approximated.

- More general theorems by Leshno *et al.* (1993) and Bach (2017).
- What about Chebyshev polynomials? Splines? Problems of convergence and generalization ("extrapolation").
- There is another, yet more subtle curse of dimensionality: data availability. We will return to this concern while dealing with symmetries

Training

Loss function

- We need to specify a loss function to train the network (i.e., select θ).
- A natural loss function: the quadratic error function $\mathcal{E}\left(\theta; \mathbf{Y}, \widehat{\mathbf{y}}\right)$:

$$\begin{aligned}
\theta^* &= \arg\min_{\theta} \mathcal{E}\left(\theta; \mathbf{Y}, \widehat{\mathbf{y}}\right) \\
&= \arg\min_{\theta} \sum_{l=1}^{L} \mathcal{E}\left(\theta; y_l, \widehat{y}_l\right) \\
&= \arg\min_{\theta} \frac{1}{2} \sum_{l=1}^{L} \left\| y_l - g\left(\mathbf{x}_l; \theta\right) \right\|^2
\end{aligned}$$

- Where from do the observations $\mathbf{Y}$ come? Observed data vs. simulated epochs.
- Initial θ come from a normal distribution $\mathcal{N}(0, \sigma)$ to break symmetry. For example $\sigma = 4\sqrt{\frac{2}{n_{input} + n_{output}}}$, but other choices are possible.

Alternative loss functions

- Other loss functions can be used.
- For instance, we can add regularization terms:
  1. ℓ_1 (LASSO): $\lambda \sum_{i=1} |\theta_i|$.
  2. ℓ_2 (ridge regression, aka as Tikhonov regularization): $\lambda \sum_{i=1} \theta_i^2$.
  3. A combination of both norms (elastic net): $\lambda_1 \sum_{i=1} |\theta_i| + \lambda_2 \sum_{i=1} \theta_i^2$.

Backpropagation

- We can easily calculate $\mathcal{E}(\theta^*; Y, \widehat{\mathbf{y}})$ and $\nabla\mathcal{E}(\theta^*; Y, \widehat{\mathbf{y}})$ for a given θ^*.
- In particular, for the gradient, we use *backpropagation* (Rumelhart *et al.*, 1986):

$$\begin{aligned}
\frac{\partial\mathcal{E}\left(\theta; y_l, \widehat{y}_l\right)}{\partial\theta_0} &= y_l - g\left(\mathbf{x}_l; \theta\right) \\
\frac{\partial\mathcal{E}\left(\theta; y_l, \widehat{y}_l\right)}{\partial\theta_m} &= \left(y_l - g\left(\mathbf{x}_l; \theta\right)\right)\phi\left(z_m\right), \text{ for } \forall m \\
\frac{\partial\mathcal{E}\left(\theta; y_l, \widehat{y}_l\right)}{\partial\theta_{n,m}} &= \left(y_l - g\left(\mathbf{x}_l; \theta\right)\right)\theta_m x_n \phi'\left(z_m\right), \text{ for } \forall n, m
\end{aligned}$$

  where $\phi'(z)$ is the derivative of the activation function.

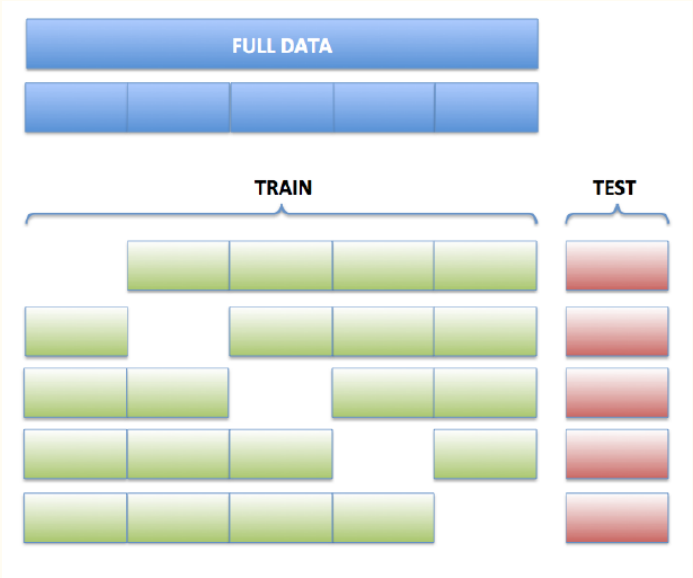
- The derivative $\phi'(z)$ will be trivial to evaluate if we use a ReLU. Also, modern libraries use automatic differentiation, which interacts particularly well with backpropagation.
- Backpropagation will be particularly important when we use multiple layers.

Architecture design

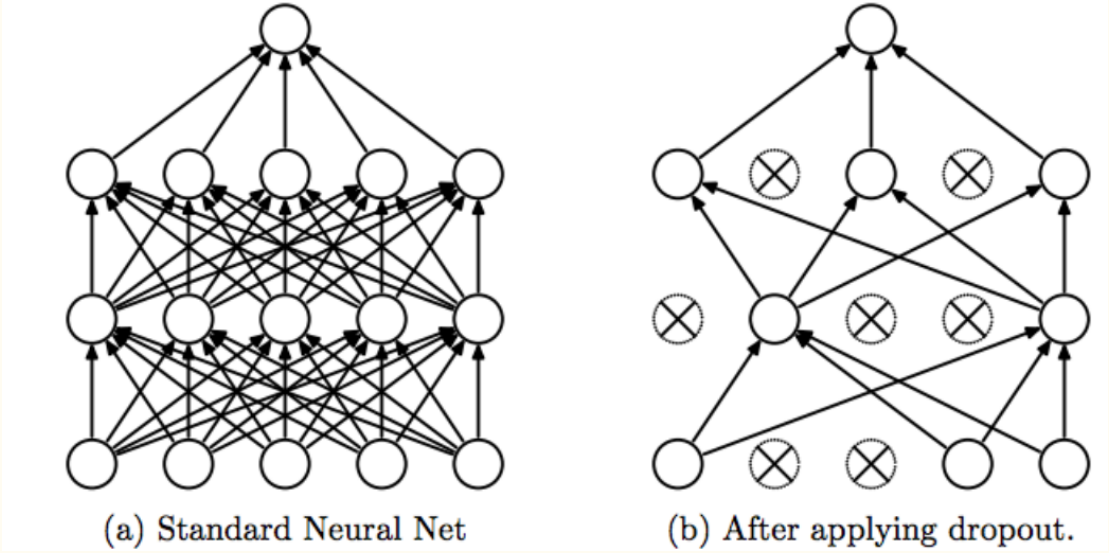
Architecture design

- Before, we have taken many aspects of the network architecture as given.
- But in practice, you need to design them (hence, importance of having access to a good deep learning library).
- Choices ("hyperparameters"):
  1. $\phi(\cdot)$: activation function.
  2. M: number of neurons.
  3. J: number of layers.
  4. Number and size of epochs.
- Notation for whole architecture: $\mathcal{A}$.
- Use $\mathcal{E}\left(\theta; \mathbf{Y}, \widehat{\mathbf{y}}\right)$ with some form of regularization (ℓ_1 or ℓ_2), cross-validation, or dropout.

Cross-validation



Dropout



(a) Standard Neural Net

(b) After applying dropout.

An online illustration

- We can play with many of these hyperparameters easily with the right libraries.
- Nothing substitutes practice.
- An interesting additional webpage: https://playground.tensorflow.org/.
- You can play with all the aspects of the architecture in several standard problems (from easy to challenging).
- Spend some time with this webpage!

Further ideas

- Principles:
  1. Trade-off error/computational time.
  2. Better to err on the side of too many M.
- Double descent phenomenon.

Generalization vs. overfitting

The classical view: Enrico Fermi, 1953

I remember my friend Johnny von Neumann used to say, with four parameters I can fit an elephant, and with five I can make him wiggle his trunk.

The modern view: Ruslan Salakhutdinov, 2017

The best way to solve the problem from practical standpoint is you build a very big system. If you remove any of these regularizations like dropout or L2, basically you want to make sure you hit the zero training error. Because if you don't, you somehow waste the capacity of the model.


Expected
(Modern ML)
Expected
(Classical Statistics)
Reality
Test / Train Error
0.7
0.6
0.5
0.4
0.3
0.2
1
10
20
30
40
50
60
Model Size

Some recent results

- Bubeck and Sellke (2021).
- They prove that for a broad class of data distributions and model classes, overparametrization is necessary if one wants to interpolate the data smoothly.
- Intuition: a tradeoff between the size of a model and its robustness.

Optimization

Descent direction iteration

- Most training of neural networks is done with (first-order) descent direction iteration methods.
- Starting at point $\theta^{(1)}$ (determined by domain knowledge), a descent direction algorithm generates sequence of steps (called iterates) that converge to a local minimum.
- The descent direction iteration algorithm:
  1. At iteration k, check whether $\theta^{(k)}$ satisfies termination condition. If so stop; otherwise go to step 2.
  2. Determine the descent direction $\mathbf{d}^{(k)}$ using local information such as gradient or Hessian.
  3. Compute step size $\alpha^{(k)}$.
  4. Compute the next candidate point: $\theta^{(k+1)} \leftarrow \theta^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}$.
- Choice of α and $\mathbf{d}$ determines the flavor of the algorithm.

Gradient descent method I

- A natural choice for $\mathbf{d}$ is the direction of steepest descent (first proposed by Cauchy in 1847).
- The direction of steepest descent is given by the direction opposite the gradient $\nabla\mathcal{E}(\theta)$. Thus, a.k.a. steepest descent.
- If function is smooth and the step size small, the method leads to improvement (as long as the gradient is not zero).
- The normalized direction of steepest descent is:

$$\mathbf{d}^{(k)} = -\frac{\nabla\mathcal{E}(\theta^{(k)})}{||\nabla\mathcal{E}(\theta^{(k)})||}$$

Gradient descent method II

- One way to set the step size is to solve a line search:

$$\alpha^k = \arg\min_{\alpha} \mathcal{E}(\theta^{(k)} + \alpha \mathbf{d}^{(k)})$$

  for example with the Brent-Dekker method.

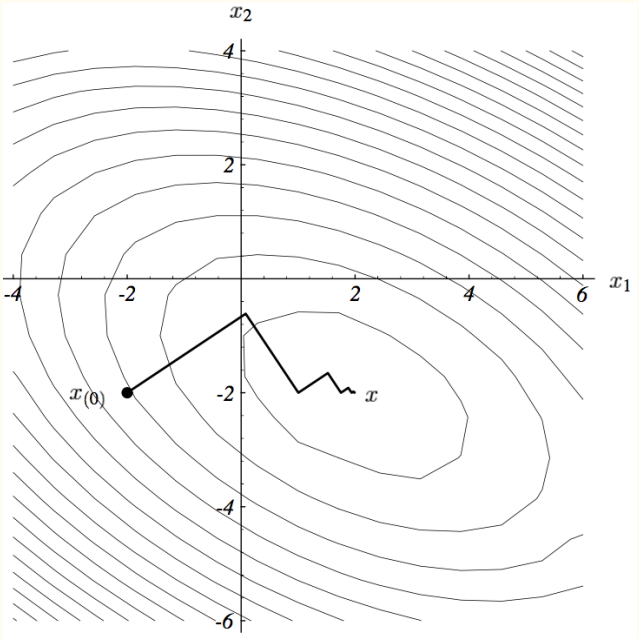
- Under this step size choice, it can be shown $\mathbf{d}^{(k+1)}$ and $\mathbf{d}^{(k)}$ are orthogonal.

- In practice, line search can be costly and we settle for a fix α, a α^k that geometrically decays, or an approximated line search.

- Trade off between speed of convergence and robustness.

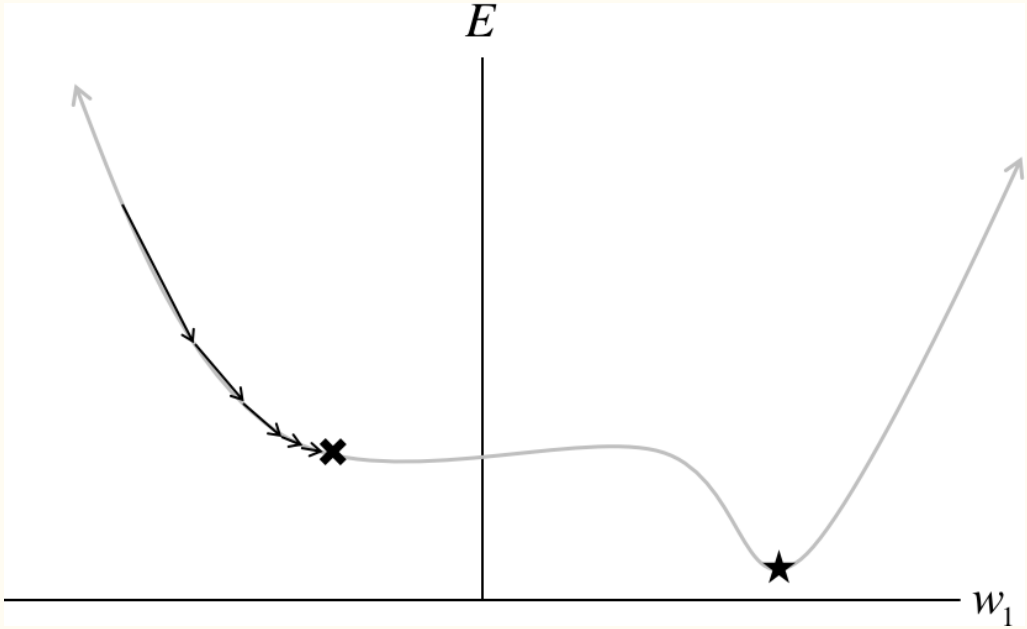
Heard in Minnesota Econ grad student lab

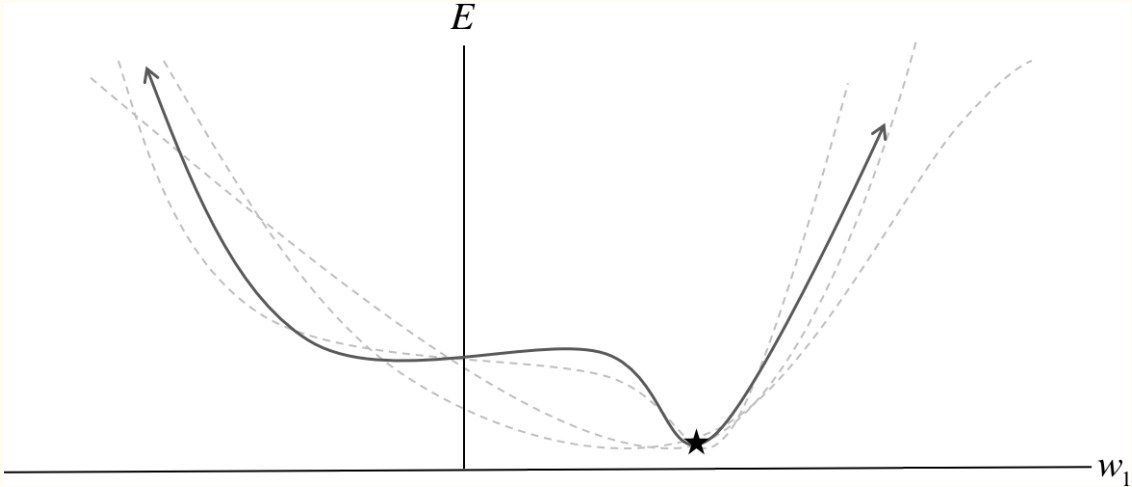
If you do not know where you are going, at least go slowly.


x_2
4
2
-4
-2
2
4
6
x_1
$x_{(0)}$
-2
x
-4
-6

Stochastic gradient descent

- Even with back propagation, evaluating the gradient for the whole training set can be costly: thousands of points to evaluate!
- Stochastic gradient descent (SGD): We use only one data point to evaluate (an approximation to) the gradient.
- We trade off slower convergence rate for faster computation and early insights in the network behavior.
- Also, noisy update process can allow the model to avoid local minima (implicit regularization).
- In practice, we do not need a global min ($\neq$ likelihood). Optimization is not an end in and of itself (also, subtle issue of non-uniqueness when models are over-parametrized).


E
w_1


E
w_1

Minibatch

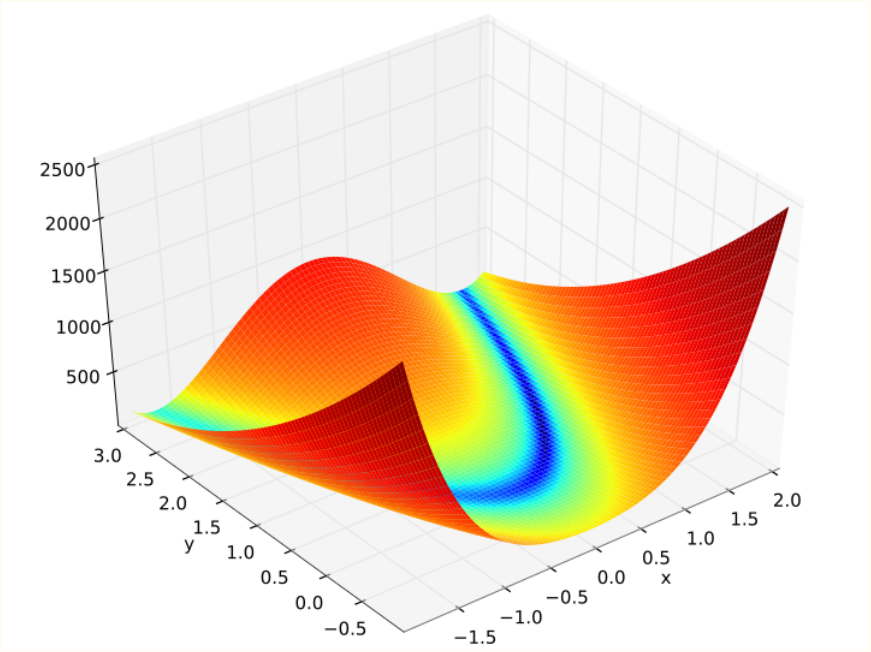
- A compromise between using the whole training set and pure stochastic gradient descent: minibatch gradient descent.
- This is the most popular algorithm to train neural networks.
- Intuition: the standard error of the mean converges slowly ($\sqrt{n}$).
- Also, usually more resilient to scaling of the update.
- Drawback: one more hyperparameter to determine.

Improving gradient descent

- Gradient descent can perform poorly in narrow valleys (it may require many steps to make progress).
- Famous example: Rosenbrock function $\rightarrow (a - x)^2 + b(y - x^2)^2$.
- Unfortunately, these are not exotica.
- If the function to minimize has flat areas, one can introduce a *momentum* update equation:

$$v^{(k+1)} = \beta v^{(k)} - \alpha g^{(k)}$$
$$\theta^{(k+1)} = \theta^{(k)} + v^{(k+1)}$$

- Application to neural network training: *Adam* (Adaptive Moment Estimation), Kingma and Ba (2014).


2500
2000
1500
1000
500
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
y
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x

Solving models in macrofinance

Functional equations

- A large class of problems in economics search for a function d that solves a *functional equation*:

$$\mathcal{H}(d) = \mathbf{0}$$

- Points to remember:

1. Regular equations are particular examples of functional equations.
2. $\mathbf{0}$ is the space zero, different in general that the zero in the reals.
3. This formalism deals both with market equilibrium and social planner problems.

Example I: decision rules

- Take the basic stochastic neoclassical growth model:

$$\max \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t)$$
$$c_t + k_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta) k_t, \ \forall \ t > 0$$
$$z_t = \rho z_{t-1} + \sigma \varepsilon_t, \ \varepsilon_t \sim \mathcal{N}(0,1)$$

- The first-order condition:

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

Example I: decision rules

- There is a decision rule (a.k.a. policy function) that gives the optimal choice of consumption and capital tomorrow given the states today:

$$d = \begin{cases} d^1(k_t, z_t) = c_t \\ d^2(k_t, z_t) = k_{t+1} \end{cases}$$

- Then:

$$\mathcal{H} = u'\left(d^1(k_t, z_t)\right)$$
$$-\beta \mathbb{E}_t \left\{ u'\left(d^1\left(d^2(k_t, z_t), z_{t+1}\right)\right) \left(1 + \alpha e^{z_{t+1}} \left(d^2(k_t, z_t)\right)^{\alpha-1} - \delta\right) \right\} = 0$$

- If we find d, and a transversality condition is satisfied, we are done!

Example II: conditional expectations

- Let us go back to our Euler equation:

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

- Define now:

$$d = \begin{cases} d^1(k_t, z_t) = c_t \\ d^2(k_t, z_t) = \mathbb{E}_t \left\{ u'(c_{t+1}) \left(1 + \alpha e^{z_{t+1}} k_{t+1}^{\alpha-1} - \delta\right) \right\} \end{cases}$$

- Why? Example: ZLB.

- Then:

$$\mathcal{H}(d) = u'\left(d^1(k_t, z_t)\right) - \beta d^2(k_t, z_t) = \mathbf{0}$$

Example III: value functions

- There is a recursive problem associated with the previous sequential problem:

$$V(k_t, z_t) = \max_{k_{t+1}} \{u(c_t) + \beta \mathbb{E}_t V(k_{t+1}, z_{t+1})\}$$

$$c_t = e^{z_t} k_t^{\alpha} + (1-\delta) k_t - k_{t+1}, \ \forall \ t > 0$$

$$z_t = \rho z_{t-1} + \sigma \varepsilon_t, \ \varepsilon_t \sim \mathcal{N}(0,1)$$

- Then:

$$d(k_t, z_t) = V(k_t, z_t)$$

and

$$\mathcal{H}(d) = d(k_t, z_t) - \max_{k_{t+1}} \{u(c_t) + \beta \mathbb{E}_t d(k_{t+1}, z_{t+1})\} = \mathbf{0}$$

Deep learning to solve functional equations

- General idea: substitute $d(\mathbf{x})$ by $d^n(\mathbf{x}, \theta)$ where θ is an $n-$dim vector of coefficients to be determined.
- We can "learn" this function with deep learning (notice: notation with just one layer):

$$d \cong d^n(\mathbf{x}; \theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi(z_m)$$

where $\phi(\cdot)$ is an arbitrary activation function and:

$$z_m = \sum_{n=0}^{N} \theta_{n,m} x_n$$

- We need to sample the desired function and minimize a loss function.

Outline

- Natural sampling: simulated paths of the economy.
- Natural loss function: the implied error in the equilibrium/optimality conditions, as loss function.
- Natural minimization algorithm: Stochastic gradient descent.
- However, you want to think about this more as a framework than a concrete set of instructions.
- Let us go over different applications. Enumeration, no exhaustive list.

Examples of code

1. An LQ optimal control problem:

   https://github.com/Mekahou/Fun-Stuff/blob/main/codes/linear%20quadratic%20DP%20DNN/3.%20LQ_DP_DNN_Training_Main.ipynb

2. A Neoclassical growth model (discrete time):

   https://colab.research.google.com/drive/1jbSti3LkxASZg04Bkod0EBJFXdf6xu9_?usp=sharing

3. A Neoclassical growth model (continuous time):

   https://colab.research.google.com/drive/1rFfqbJYn26_nm-CS21Fbw_QV7ccM8Xl1T?usp=sharing

4. An OLG model:

   https://github.com/sischei/DeepEquilibriumNets

5. A Krusell-Smith and a financial frictions HA code:

   https://github.com/jesusfv/financial-frictions

Dynamic programming

Solving high-dimensional dynamic programming problems (discrete time)

- Our goal is to solve the Bellman equation globally:

$$
\begin{aligned}
V(\mathbf{x}) &= \max_{\boldsymbol{\alpha}} r(\mathbf{x}, \boldsymbol{\alpha}) + \beta * \mathbb{E}V(\mathbf{x}') \\
\text{s.t. } \mathbf{x}' &= f(\mathbf{x}, \boldsymbol{\alpha}) \\
G(\mathbf{x}, \boldsymbol{\alpha}) &\leq \mathbf{0} \\
H(\mathbf{x}, \boldsymbol{\alpha}) &= \mathbf{0}
\end{aligned}
$$

- Notice that framework is very general. Usually we will not have all these constraints.
- Think about the cases where we have many state variables. Trade-offs in terms of speed vs. scalability.

Solving high-dimensional dynamic programming problems (continuous time)

- We can also write the equivalent continuous-time Hamilton-Jacobi-Bellman (HJB) equation globally:

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

- Solution algorithm is absolutely the same.
- When discrete vs. continuous?

Neural networks

- We define four neural networks:
  1. $\tilde{V}(\mathbf{x}; \boldsymbol{\Theta}^V) : \mathbb{R}^N \to \mathbb{R}$ to approximate the value function $V(\mathbf{x})$.
  2. $\tilde{\alpha}(\mathbf{x}; \boldsymbol{\Theta}^\alpha) : \mathbb{R}^N \to \mathbb{R}^P$ to approximate the policy function $\boldsymbol{\alpha}$.
  3. $\tilde{\mu}(\mathbf{x}; \boldsymbol{\Theta}^\mu) : \mathbb{R}^N \to \mathbb{R}^{L_1}$, and $\tilde{\lambda}(\mathbf{x}; \boldsymbol{\Theta}^\lambda) : \mathbb{R}^N \to \mathbb{R}^{L_2}$ to approximate the Karush-Kuhn-Tucker (KKT) multipliers $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$.
- To simplify notation, we accumulate all weights in the matrix $\boldsymbol{\Theta} = (\boldsymbol{\Theta}^V, \boldsymbol{\Theta}^\alpha, \boldsymbol{\Theta}^\mu, \boldsymbol{\Theta}^\lambda)$.
- We could think about the approach as just one large neural network with multiple outputs.

Error criterion I (discrete time)

- The Bellman error:

$$err_B(\mathbf{x};\boldsymbol{\Theta}) \equiv r(\mathbf{x},\tilde{\alpha}(\mathbf{s};\boldsymbol{\Theta}^{\alpha})) + \beta * \mathbb{E}\tilde{V}(\mathbf{x}';\boldsymbol{\Theta}^{V}) - \tilde{V}(\mathbf{x};\boldsymbol{\Theta}^{V})$$

- The policy function error:

$$err_{\alpha}(\mathbf{x};\boldsymbol{\Theta}) \equiv \frac{\partial r(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^{\alpha})}{\partial \boldsymbol{\alpha}} + \beta * D_{\alpha}f(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^{\alpha}))^{T}\mathbb{E}\nabla_{x}\tilde{V}(\mathbf{x}';\boldsymbol{\Theta}^{V}) - D_{\alpha}G(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^{\alpha}))^{T}\tilde{\mu}(\mathbf{x};\boldsymbol{\Theta}^{\mu}) - D_{\alpha}H(\mathbf{x},\tilde{\alpha}(\mathbf{x};\boldsymbol{\Theta}^{\alpha}))\tilde{\lambda}(\mathbf{x};\boldsymbol{\Theta}^{\lambda}),$$

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

Error criterion I (continuous time)

- The HJB error:

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

- The policy function error:

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

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

Error criterion II (discrete and continuous time)

- The constraint error is itself composed of the primal feasibility errors:

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

  the dual feasibility error:

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

  and the complementary slackness error:

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

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

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

Training

- We train our neural networks by minimizing the above error criterion through minibatch gradient descent over points drawn from the ergodic distribution of the state vector.
- We initialize our network weights and perform K learning steps called epochs, where K can be chosen in a variety of ways.
- For each epoch, we draw I points from the state space by simulating from the ergodic distribution.
- Then, we randomly split this sample into B minibatches of size S. For each minibatch, we define the minibatch error,by averaging the loss function over the batch.
- Finally, we perform minibatch gradient descent for all network weights, with η_k being the learning rate in the k-th epoch.

The continuous-time neoclassical growth model

- Standard problem in economics.
- A single agent deciding to either save in capital or consume with a HJB equation :

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

- Notice that $c = (U')^{-1}(V'(k))$. With CRRA utility, this simplifies further to $c = (V'(k))^{-\frac{1}{\gamma}}$.
- We set $\gamma = 2$, $\rho = 0.04$, $F(k) = 0.5 * k^{0.36}$, $\delta = 0.05$.

Approximating the value function

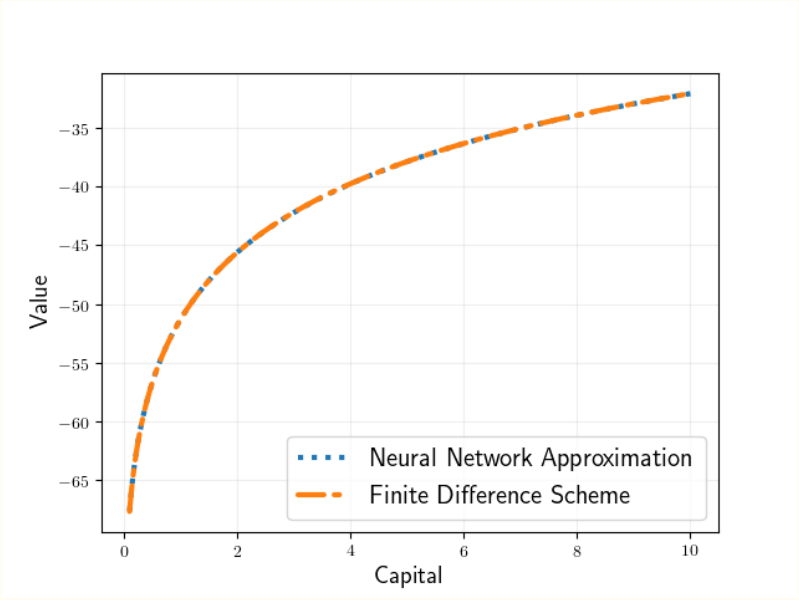
- First, we approximate the value function $V(k)$ with a neural network, $\tilde{V}(k;\Theta)$ and "HJB error":

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

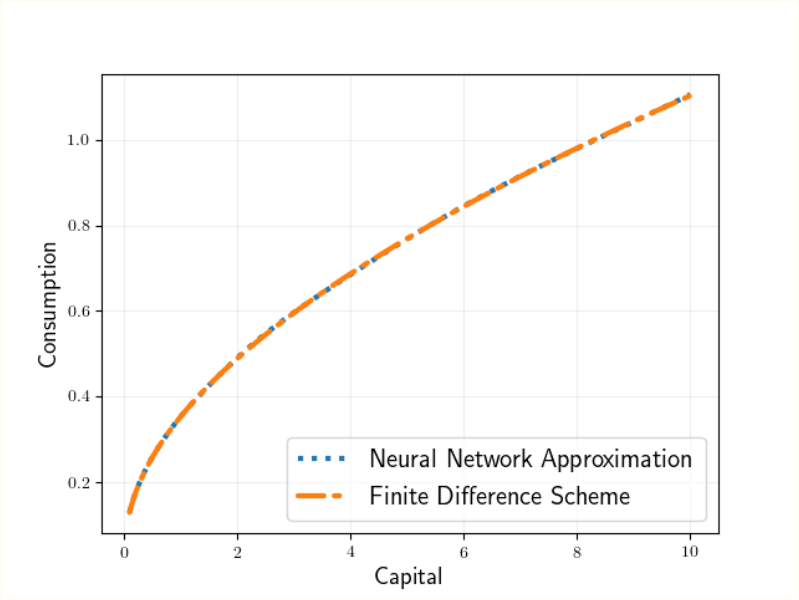
  using the known functional form of the policy function.

- Details:

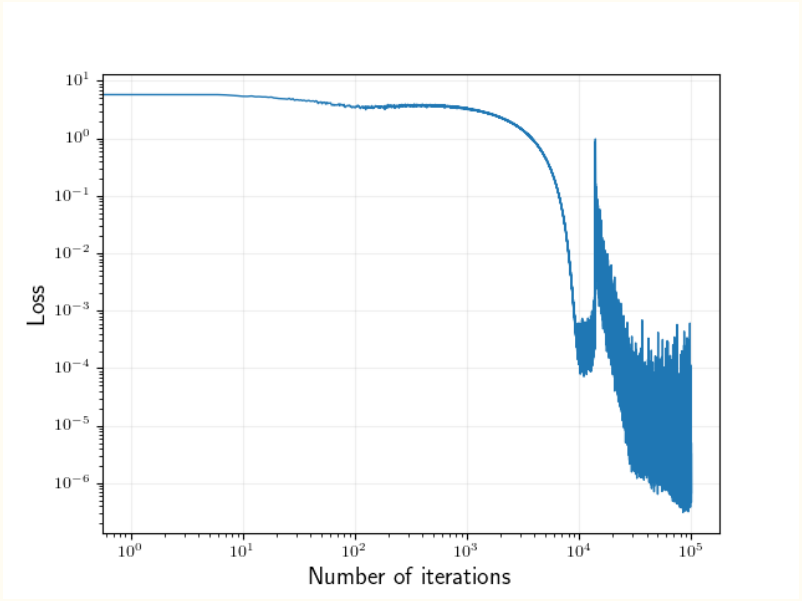
  1. 3 layers with 8 neurons per layer.
  2. $\tanh(x)$ activation.
  3. Gaussian initialization $\mathcal{N}\left(0, 4\sqrt{\frac{2}{n_{input}+n_{output}}}\right)$ with input normalization.



(a) Value with closed-form policy



(c) Consumption with closed-form policy



(e) HJB error with closed-form policy

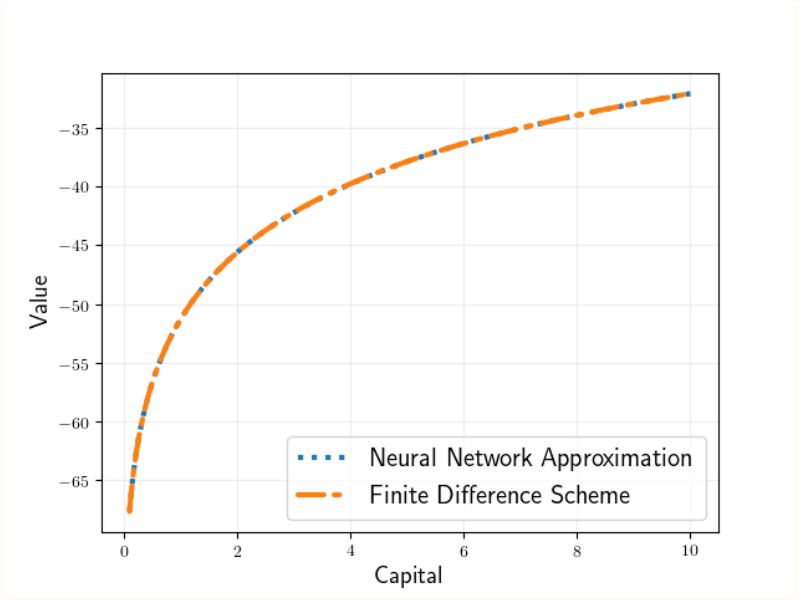
Approximating the policy function

- Let us now approximate the policy function as well with a policy neural network $\tilde{C}(k;\Theta^C)$.
- The new HJB error:

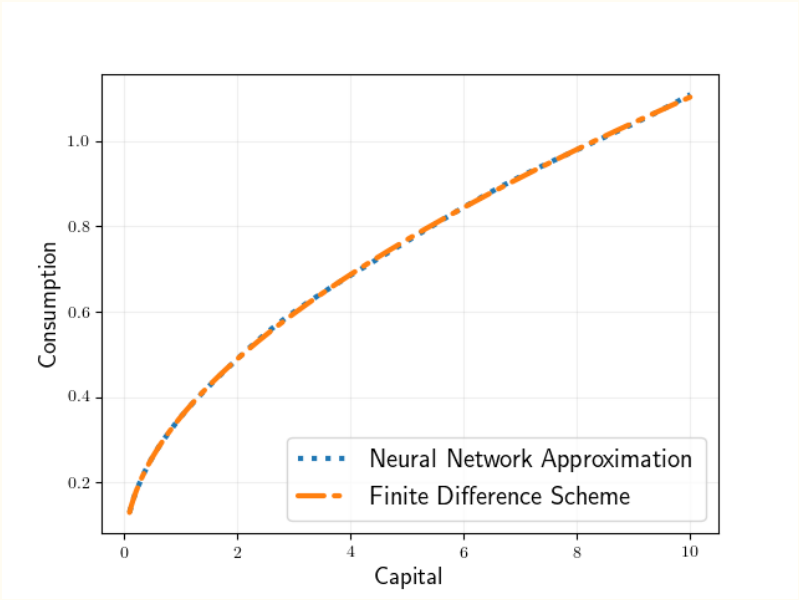
$$err_{HJB} = \rho\tilde{V}(k;\Theta^V) - U\left(\tilde{C}(k;\Theta^C)\right) - \frac{\partial\tilde{V}(k;\Theta^V)}{\partial k}\left[F(k) - \delta * k - \tilde{C}(k;\Theta^C)\right]$$

- Now we also have a policy function error:

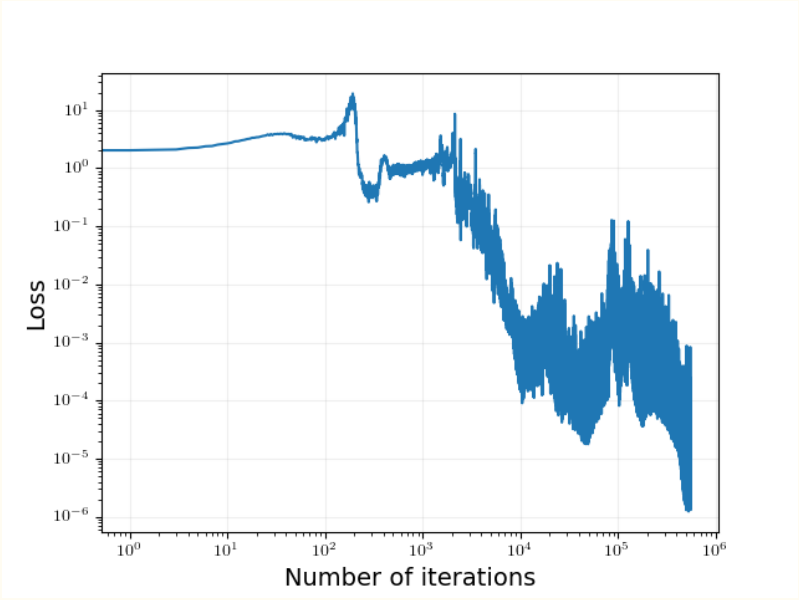
$$err_C = (U')^{-1}\left(\frac{\partial\tilde{V}(k;\Theta^V)}{\partial k}\right) - \tilde{C}(k;\Theta^C)$$



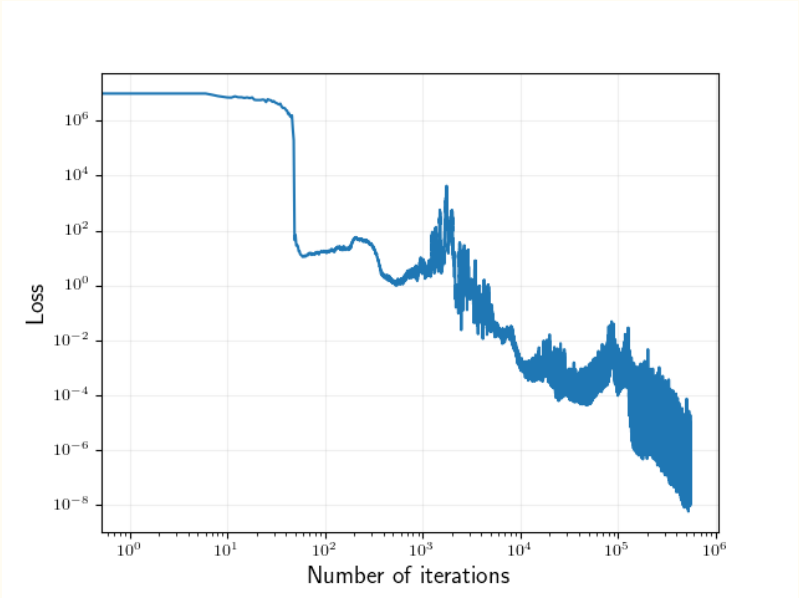
(b) Value with policy approximation



(d) Consumption with policy approximation



(f) HJB error with policy approximation



(g) Policy error with policy approximation

Models with heterogeneous agents

The challenge

- To compute and take to the data models with heterogeneous agents, we need to deal with:
  1. The distribution of agents G_t.
  2. The operator $H(\cdot)$ that characterizes how G_t evolves:

     $$G_{t+1} = H(G_t, S_t)$$

     or

     $$\frac{\partial G_t}{\partial t} = H(G_t, S_t)$$

     given the other aggregate states of the economy S_t.
- How do we track G_t and compute $H(G_t, S_t)$?

A common approach

- If we are dealing with N discrete types, we keep track of $N-1$ weights.
- If we are dealing with continuous types, we extract a finite number of features from G_t:
  1. Moments.
  2. Q-quantiles.
  3. Weights in a mixture of normals...
- We stack either the weights or features of the distribution in a vector μ_t.
- We assume μ_t follows the operator $h(\mu_t, S_t)$ instead of $H(G_t, S_t)$.
- We parametrize $h(\mu_t, S_t)$ as $h^j(\mu_t, S_t; \theta)$.
- We determine the unknown coefficients θ such that an economy where μ_t follows $h^j(\mu_t, S_t; \theta)$ replicates as well as possible the behavior an economy where G_t follows $H(\cdot)$.

Example: Basic Krusell-Smith model

- Two aggregate variables: aggregate productivity shock and household distribution $G_t(a, z)$ where:

$$\int G_t(a, z)da = K_t$$

- We summarize $G_t(a, \cdot)$ with the log of its mean: $\mu_t = \log K_t$ (extending to higher moments is simple, but tedious).

- We parametrize $\underbrace{\log K_{t+1}}_{\mu_{t+1}} = \underbrace{\theta_0(s_t) + \theta_1(s_t) \log K_t}_{h^j(\mu_t, s_t; \theta)}$.

- We determine $\{\theta_0(s_t), \theta_1(s_t)\}$ by OLS run on a simulation.

Problems

- No much guidance regarding feature and parameterization selection in general cases.
  - Yes, keeping track of the log of the mean and a linear functional form work well for the basic model. But, what about an arbitrary model?
  - Method suffers from "curse of dimensionality": difficult to implement with many state variables or high N/higher moments.
  - Lack of theoretical foundations (Does it converge? Under which metric?).

How can deep learning help?

- Deep learning addresses challenges:
  1. How to extract features from an infinite-dimensional object efficiently.
  2. How to parametrize the non-linear operator mapping how distributions evolve.
  3. How to tackle the "curse of dimensionality."
- Given time limitations, today I will discuss the last two points.
- In our notation of $y = f(\mathbf{x})$:
  1. $y = \mu_{t+1}$.
  2. $\mathbf{x} = (\mu_t, S_t)$.

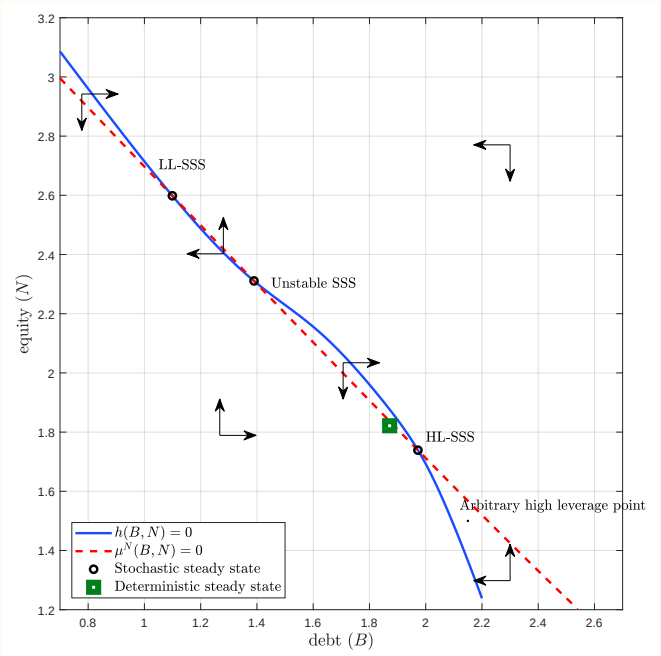
Example: Financial frictions and the wealth distribution

Motivation

- Recently, many papers have documented the nonlinear relations between financial variables and aggregate fluctuations.
- For example, Jordà *et al.* (2016) have gathered data from 17 advanced economies over 150 years to show how output growth, volatility, skewness, and tail events all seem to depend on the levels of leverage in an economy.
- Similarly, Adrian *et al.* (2019a) have found how, in the U.S., sharply negative output growth follows worsening financial conditions associated with leverage.
- Can a fully nonlinear DSGE model account for these observations?
- To answer this question, we postulate, compute, and estimate a continuous-time DSGE model with a financial sector, modeled as a representative financial expert, and households, subject to uninsurable idiosyncratic labor productivity shocks.

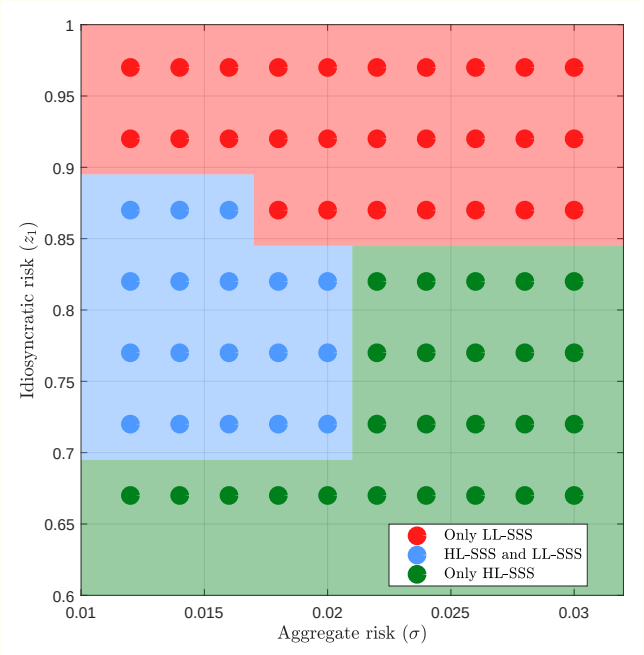
The main takeaway

- The interaction between the supply of bonds by the financial sector and the precautionary demand for bonds by households produces significant *endogenous aggregate risk*.
- This risk induces an endogenous regime-switching process for output, the risk-free rate, excess returns, debt, and leverage.
- Mechanism: *endogenous aggregate risk* begets multiple stochastic steady states or SSS(s), each with its own stable basin of attraction.
- Intuition: different persistence of wages and risk-free rates in each basin.
- The regime-switching generates:
  1. Multimodal distributions of aggregate variables.
  2. Time-varying levels of volatility and skewness for aggregate variables.
  3. Supercycles of borrowing and deleveraging.


LL-SSS
Unstable SSS
HL-SSS
Arbitrary high leverage point
$h(B,N)=0$
$\mu^N(B,N)=0$
Stochastic steady state
Deterministic steady state
equity (N)
debt (B)
3.2
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6

HA vs. RA

- Our findings are in contrast with the properties of the representative household version of the model.
- While the consumption decision rule of the households is close to linear with respect to the household state variables, it is sharply nonlinear with respect to the aggregate state variables.
- This point is more general: agent heterogeneity might matter even if the decision rules of the agents are linear with respect to individual state variables.
- Thus, changes in the forces behind precautionary savings affect aggregate variables.


1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
Idiosyncratic risk (z_1)
0.01
0.015
0.02
0.025
0.03
Aggregate risk (σ)
Only LL-SSS
HL-SSS and LL-SSS
Only HL-SSS

The firm

- Representative firm with technology:

$$Y_t = K_t^{\alpha} L_t^{1-\alpha}$$

- Competitive input markets:

$$w_t = (1-\alpha) K_t^{\alpha} L_t^{-\alpha}$$
$$rc_t = \alpha K_t^{\alpha-1} L_t^{1-\alpha}$$

- Aggregate capital evolves:

$$\frac{dK_t}{K_t} = (\iota_t - \delta)\, dt + \sigma dZ_t$$

- Instantaneous return rate on capital dr_t^k:

$$dr_t^k = (rc_t - \delta)\, dt + \sigma dZ_t$$

The expert

- Representative expert (financial intermediary) with preferences:

$$\mathbb{E}_0 \left[\int_0^\infty e^{-\widehat{\rho} t} \log(\widehat{C}_t) dt \right]$$

- Expert rents capital K_t to firms and issues risk-free debt B_t at rate r_t to households.
- Financial friction: expert cannot issue state-contingent claims and must absorb all risk from capital.
- Expert's net wealth $N_t = K_t - B_t$ evolves:

$$\begin{aligned} dN_t &= K_t dr_t^k - r_t B_t dt - \widehat{C}_t dt \\ &= \left[\left(r_t + \omega_t \left(rc_t - \delta - r_t \right) \right) N_t - \widehat{C}_t \right] dt + \sigma \omega_t N_t dZ_t \end{aligned}$$

  where $\omega_t \equiv \frac{K_t}{N_t}$ is the leverage ratio.

- K_t follows:

$$dK_t = dN_t + dB_t$$

Households I

- Continuum of infinitely-lived households with unit mass with preferences:

$$\mathbb{E}_0 \left[\int_0^\infty e^{-\rho t} \frac{c_t^{1-\gamma} - 1}{1-\gamma} dt \right]$$

- Heterogeneous in wealth a_m and labor productivity z_m for $m \in [0, 1]$.

- z_t units of labor valued at wage w_t evolves stochastically following a Markov chain:

  1. $z_t \in \{z_1, z_2\}$, with $z_1 < z_2$ and ergodic mean 1.
  2. Jump intensity from state 1 to state 2: λ_1 (reverse intensity is λ_2).

- Distribution of households $G_t(a, z)$.

Households II

- Households save $a_t \geq 0$ in the riskless debt issued by experts with an interest rate r_t:

$$da_t = (w_t z_t + r_t a_t - c_t)\, dt = s\,(a_t, z_t, K_t, G_t)\, dt$$

- Optimal choice: $c_t = c\,(a_t, z_t, K_t, G_t)$.

- Total consumption by households:

$$C_t \equiv \int c\,(a_t, z_t, K_t, G_t)\, dG_t\,(a, z)$$

- $G_t\,(a, z)$ has a Radon-Nikodym derivative $g_t(a, z)$ that follows the KF equation:

$$\frac{\partial g_{it}}{\partial t} = -\frac{\partial}{\partial a}\left(s\,(a_t, z_t, K_t, G_t)\, g_{it}(a)\right) - \lambda_i g_{it}(a) + \lambda_j g_{jt}(a), \;\; i \neq j = 1, 2$$

  where $g_{it}(a) \equiv g_t(a, z_i)$, $i = 1, 2$.

Market clearing

1. Total amount of labor rented by the firm is equal to labor supplied:

$$L_t = \int z dG_t = 1$$

   Then, total payments to labor are given by w_t.

2. Total amount of debt of the expert equals the total households' savings:

$$B_t \equiv \int a dG_t (da, dz)$$

   with $dB_t = (w_t + r_t B_t - C_t) dt$.

3. By the resource constraint, $dK_t = \left(Y_t - \delta K_t - C_t - \widehat{C}_t\right) dt + \sigma K_t dZ_t$, we get:

$$\iota_t = \frac{Y_t - C_t - \widehat{C}_t}{K_t}$$

Characterizing the equilibrium I

- First, we proceed with the expert's problem. Because of log-utility:

$$\widehat{C}_t = \widehat{\rho} N_t$$

$$\omega_t = \frac{rc_t - \delta - r_t}{\sigma^2}$$

- We can use the equilibrium values of rc_t, L_t, and ω_t to get the wage:

$$w_t = (1 - \alpha) K_t^{\alpha}$$

the rental rate of capital:

$$rc_t = \alpha K_t^{\alpha - 1}$$

and the risk-free interest rate:

$$r_t = \alpha K_t^{\alpha - 1} - \delta - \sigma^2 \frac{K_t}{N_t}$$

Characterizing the equilibrium II

- Expert's net wealth evolves as:

$$dN_t = \underbrace{\left(\alpha K_t^{\alpha-1} - \delta - \widehat{\rho} - \sigma^2\left(1 - \frac{K_t}{N_t}\right)\frac{K_t}{N_t}\right) N_t dt}_{\mu_t^N(B_t,N_t)} + \underbrace{\sigma K_t}_{\sigma_t^N(B_t,N_t)} dZ_t$$

- And debt as:

$$dB_t = \left((1-\alpha)\, K_t^{\alpha} + \left(\alpha K_t^{\alpha-1} - \delta - \sigma^2 \frac{K_t}{N_t}\right) B_t - C_t\right) dt = h(g_t(a,z), N_t)dt$$

- $h(g_t(a,z), N_t)dt$ depends on:

$$C_t \equiv \int c\left(a_t, z_t, K_t, G_t\right) g_t\left(a,z\right) dadz$$

$$\frac{\partial g_{it}}{\partial t} = -\frac{\partial}{\partial a}\left(s\left(a_t, z_t, K_t, G_t\right) g_{it}(a)\right) - \lambda_i g_{it}(a) + \lambda_j g_{jt}(a), \;\; i \neq j = 1,2$$

How do we find $h(\cdot)$? Four steps

1. We substitute $h(g_t(a,z),N_t)dt$ for $h(B_t,N_t)dt$ à la Krusell and Smith (1998). The solution can be trivially extended to higher moments or q-quantiles.

2. With $h(B_t,N_t)dt$, we solve the HJB of the households.

3. We approximate $h(B_t,N_t)dt$ using a neural network:

   - Neural networks are i) universal nonlinear approximators and ii) break the curse of dimensionality.

4. With $h(B_t,N_t)dt$, we find the likelihood function of the model and perform structural estimation:

   - The operator in the KF equation that defines the likelihood function of the model is the adjoint of the infinitesimal generator of the HJB given by $h(B_t,N_t)dt$.

Solving the HJB equation

- Given $h(B_t, N_t)dt$, the household's Hamilton-Jacobi-Bellman (HJB) equation becomes:

$$\begin{aligned}\rho V_i(a,B,N) &= \max_c \frac{c^{1-\gamma}-1}{1-\gamma} + s\frac{\partial V_i}{\partial a} + \lambda_i\left[V_j(a,B,N) - V_i(a,B,N)\right] \\ &\quad + h(B,N)\frac{\partial V_i}{\partial B} + \mu^N(B,N)\frac{\partial V_i}{\partial N} + \frac{\left[\sigma^N(B,N)\right]^2}{2}\frac{\partial^2 V_i}{\partial N^2}\end{aligned}$$

  $i \neq j = 1, 2$, and where

$$s = s(a, z, N + B, G)$$

- We solve the HJB with a first-order, implicit upwind scheme in a finite difference stencil.

- Deep learning alternative: Fernández-Villaverde *et al.* (2022).

The algorithm

1) Start with h_0, an initial guess for h.

2) Using current guess h_n, solve for the household consumption, c_m, in the HJB equation.

3) Construct a time series for B_t by simulating by J periods the cross-sectional distribution of households with a constant time step Δt (starting at DSS and with a burn-in).

4) Given B_t, find N_t, K_t, and:

$$\widehat{\mathbf{h}} = \left\{ \widehat{h}_1, \widehat{h}_2 ..., \widehat{h}_j \equiv \frac{B_{t_j+\Delta t} - B_{t_j}}{\Delta t}, ..., \widehat{h}_J \right\}$$

5) Define $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_J\}$, where $\mathbf{x}_j = \left\{x_j^1, x_j^2\right\} = \left\{B_{t_j}, N_{t_j}\right\}$.

6) Use $\left(\widehat{\mathbf{h}}, \mathbf{X}\right)$ and a universal nonlinear approximator to obtain h_{n+1}, a new guess for h.

7) Iterate steps 2)-6) until h_{n+1} is sufficiently close to h_n.

A universal nonlinear approximator

- We approximate $h(\cdot)$ with a neural network (NN):

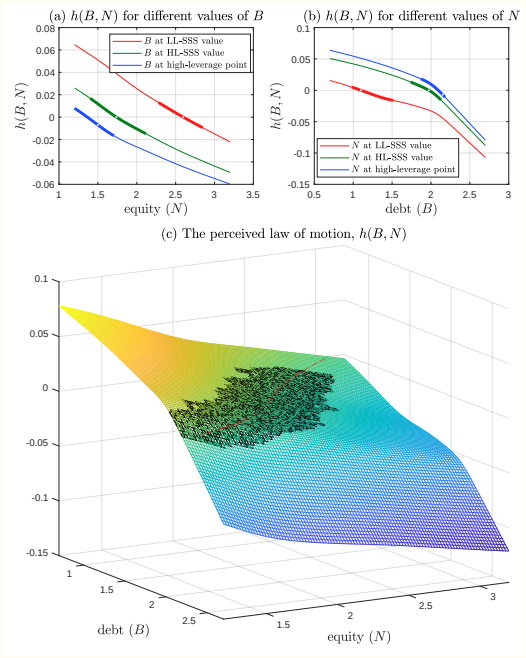
$$h(\mathbf{x};\theta) = \theta_0^1 + \sum_{q=1}^{Q} \theta_q^1 \phi\left(\theta_{0,q}^2 + \sum_{i=1}^{D} \theta_{i,q}^2 x^i\right)$$

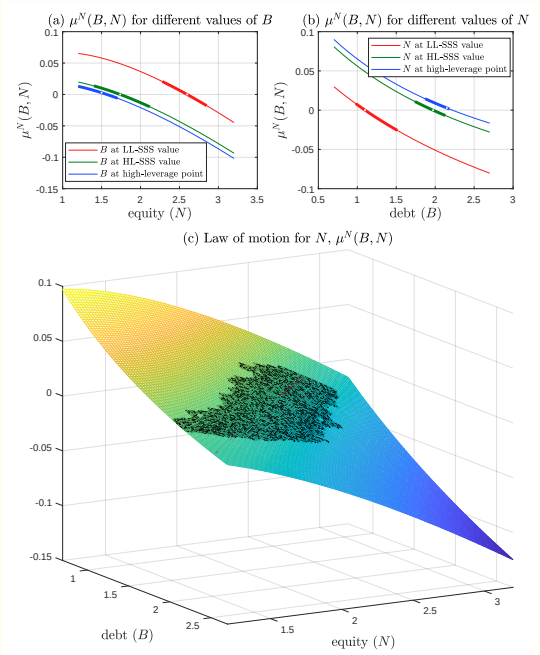
  where $Q = 16$, $D = 2$, and $\phi(x) = \log(1 + e^x)$.

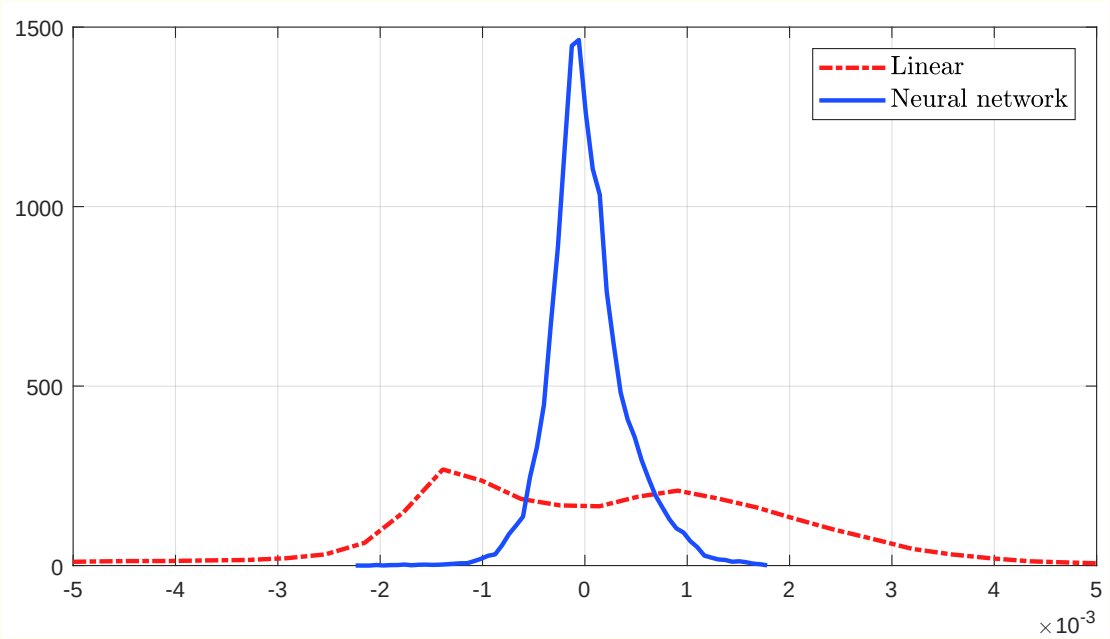
- We find:

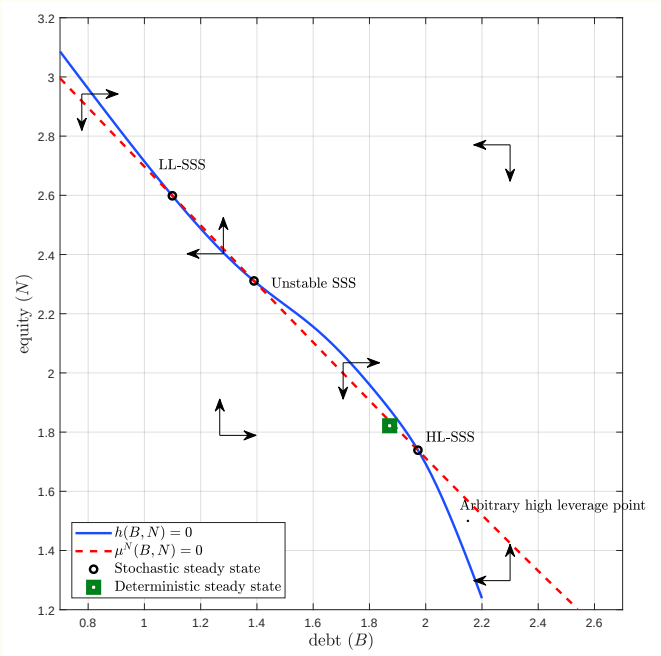
$$\theta^* = \arg\min_{\theta} \frac{1}{2} \sum_{j=1}^{J} \left\| h(\mathbf{x}_j;\theta) - \widehat{h}_j \right\|^2$$

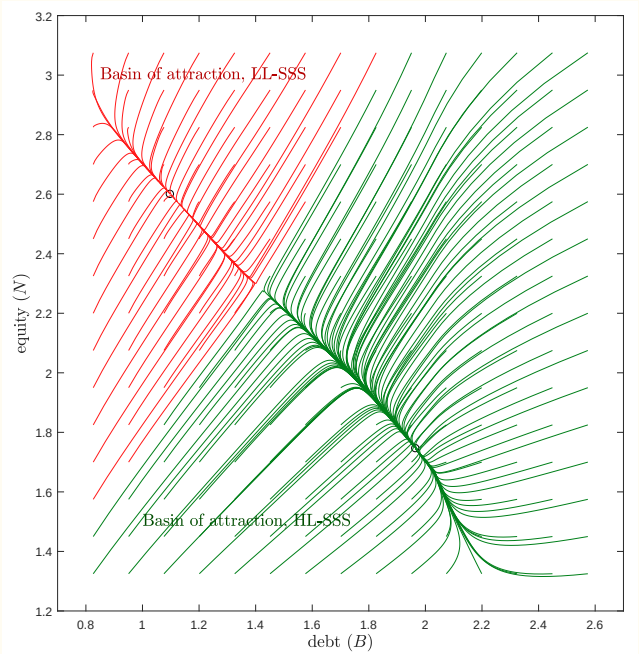
- With this solution, evaluating the likelihood function of the model is straightforward (the operator in the KF equation is the adjoint of the infinitesimal generator of the HJB).

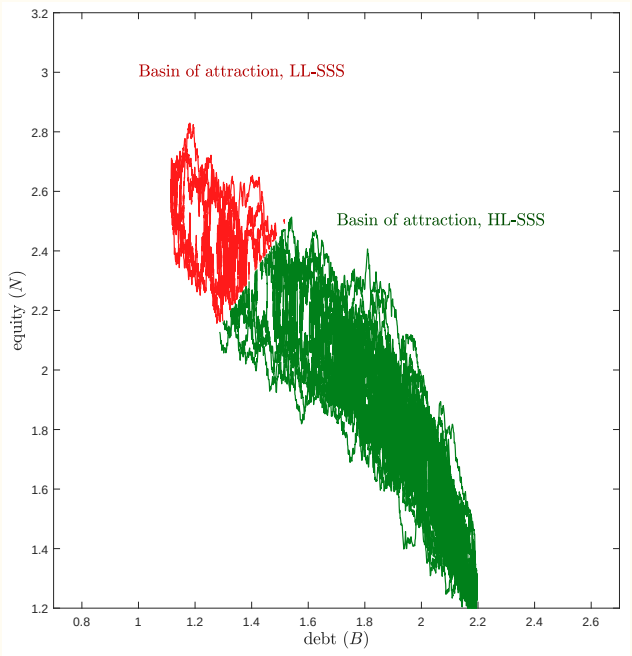

(a) $h(B,N)$ for different values of B
B at LL-SSS value
B at HL-SSS value
B at high-leverage point
$h(B,N)$
equity (N)
(b) $h(B,N)$ for different values of N
N at LL-SSS value
N at HL-SSS value
N at high-leverage point
$h(B,N)$
debt (B)
(c) The perceived law of motion, $h(B,N)$
debt (B)
equity (N)

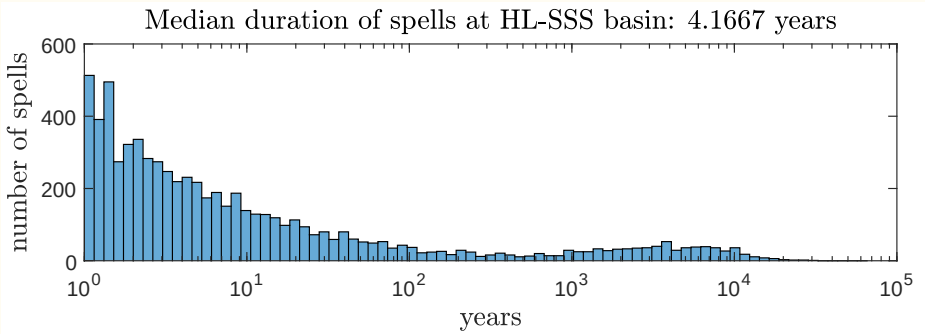

(a) $\mu^N(B,N)$ for different values of B
$\mu^N(B,N)$
equity (N)
B at LL-SSS value
B at HL-SSS value
B at high-leverage point
(b) $\mu^N(B,N)$ for different values of N
$\mu^N(B,N)$
debt (B)
N at LL-SSS value
N at HL-SSS value
N at high-leverage point
(c) Law of motion for N, $\mu^N(B,N)$
debt (B)
equity (N)

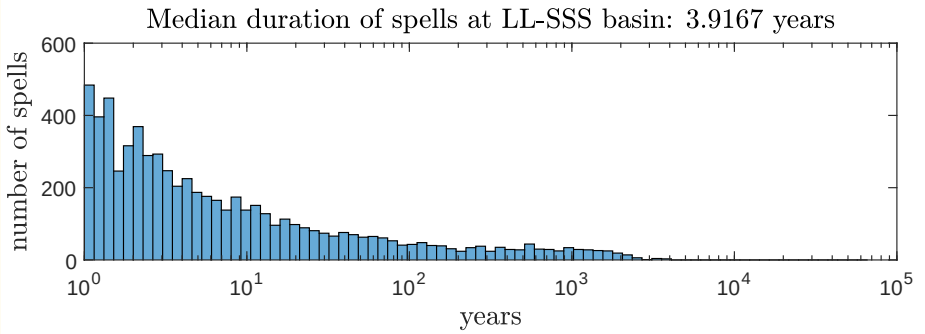

1500
1000
500
0
-5
-4
-3
-2
-1
0
1
2
3
4
5
$\times 10^{-3}$
Linear
Neural network

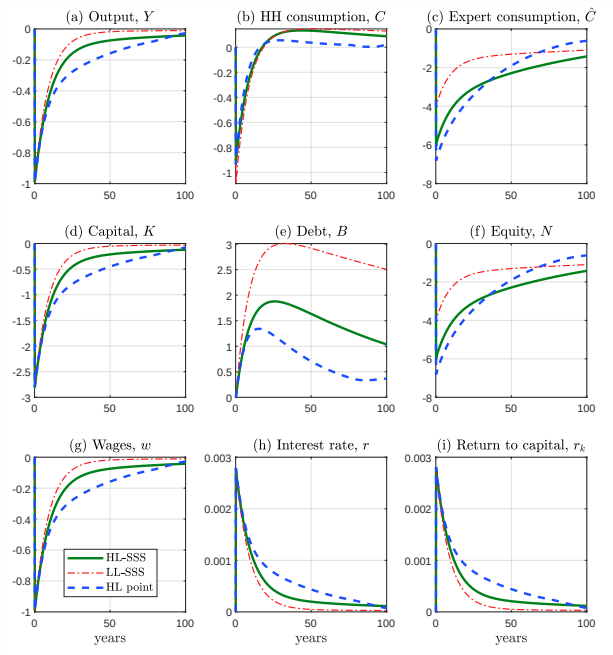

LL-SSS
Unstable SSS
HL-SSS
Arbitrary high leverage point
$h(B,N)=0$
$\mu^N(B,N)=0$
Stochastic steady state
Deterministic steady state
equity (N)
debt (B)
3.2
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6

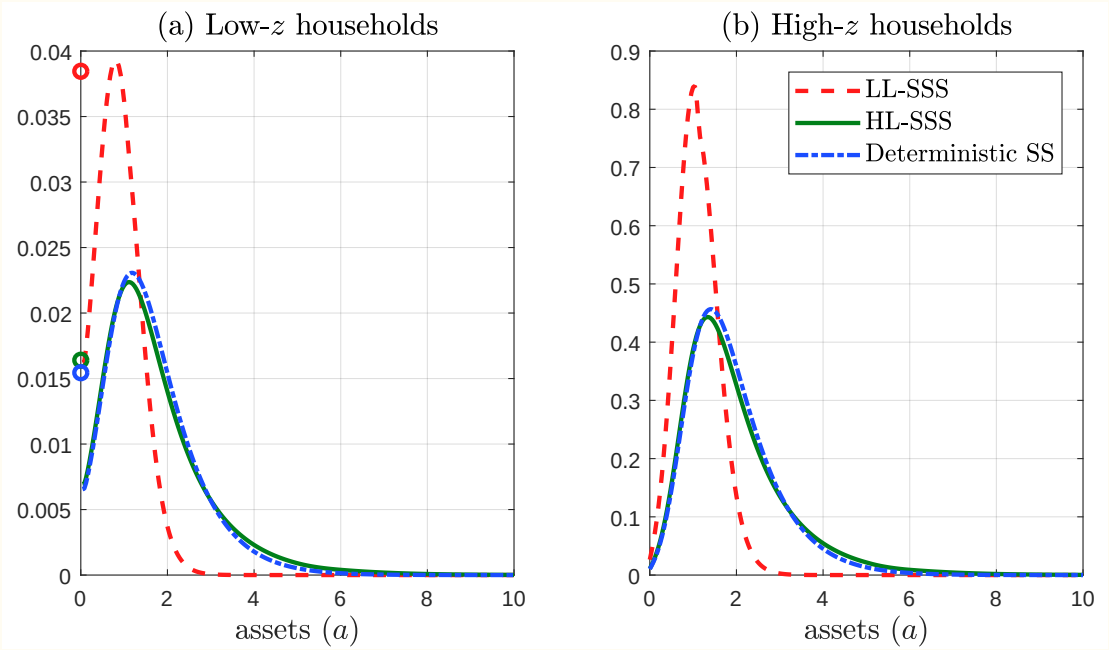

Basin of attraction, LL-SSS
Basin of attraction, HL-SSS
equity (N)
debt (B)
3.2
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6

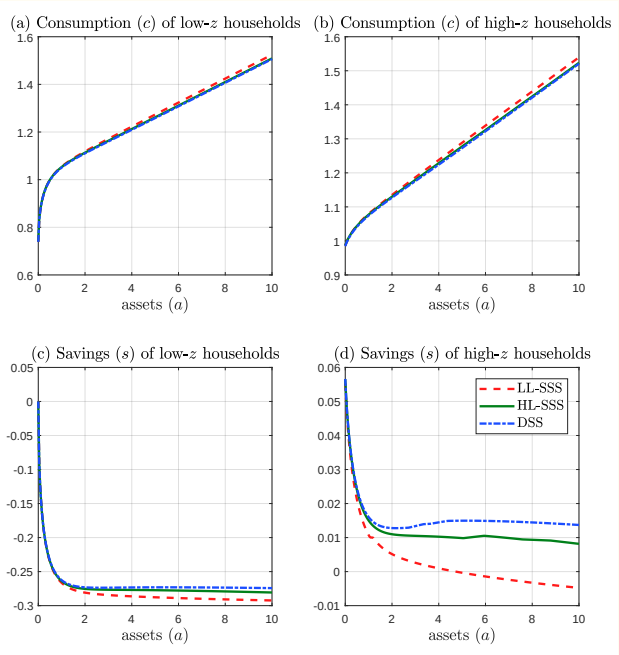

Basin of attraction, LL-SSS
Basin of attraction, HL-SSS
equity (N)
debt (B)
3.2
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6


Median duration of spells at HL-SSS basin: 4.1667 years
number of spells
600
400
200
0
10^0
10^1
10^2
10^3
10^4
10^5
years


Median duration of spells at LL-SSS basin: 3.9167 years
number of spells
600
400
200
0
10^0
10^1
10^2
10^3
10^4
10^5
years


(a) Output, Y
(b) HH consumption, C
(c) Expert consumption, $\hat{C}$
(d) Capital, K
(e) Debt, B
(f) Equity, N
(g) Wages, w
(h) Interest rate, r
(i) Return to capital, r_k
HL-SSS
LL-SSS
HL point
years
years
years


(a) Low-z households
(b) High-z households
LL-SSS
HL-SSS
Deterministic SS
assets (a)
assets (a)


(a) Consumption (c) of low-z households
assets (a)
(b) Consumption (c) of high-z households
assets (a)
(c) Savings (s) of low-z households
assets (a)
(d) Savings (s) of high-z households
LL-SSS
HL-SSS
DSS
assets (a)