INTRODUCTION TO DEEP EQUILIBRIUM NETS* & DEEP STRUCTURAL ESTIMATION**

DSE SUMMER SCHOOL 2021

Simon Scheidegger – simon.scheidegger@unil.ch

University of Lausanne, Department of Economics

August 22nd, 2021

*DEQN: joint work with M. Azinovic (UZH), L. Gaegauf (UZH)
https://github.com/sischei/DeepEquilibriumNets

**DSE: joint work with A. Didisheim (UNIL), H. Chen (MIT)
https://github.com/DeepStructuralEstimation/OptionPricing

FEW WORDS ABOUT MYSELF

- Prof. of advanced data analytics at the University of Lausanne
- Research interest in computational economics and finance, HPC, and ML applied to those fields
- https://sites.google.com/site/simonscheidegger

ROAD-MAP

- Lecture content
  - Motivation
  - A brief recap on Machine Learning Basics
  - Deep Learning Basics
  - Deep Equilibrium Nets
  - Deep Structural Estimation

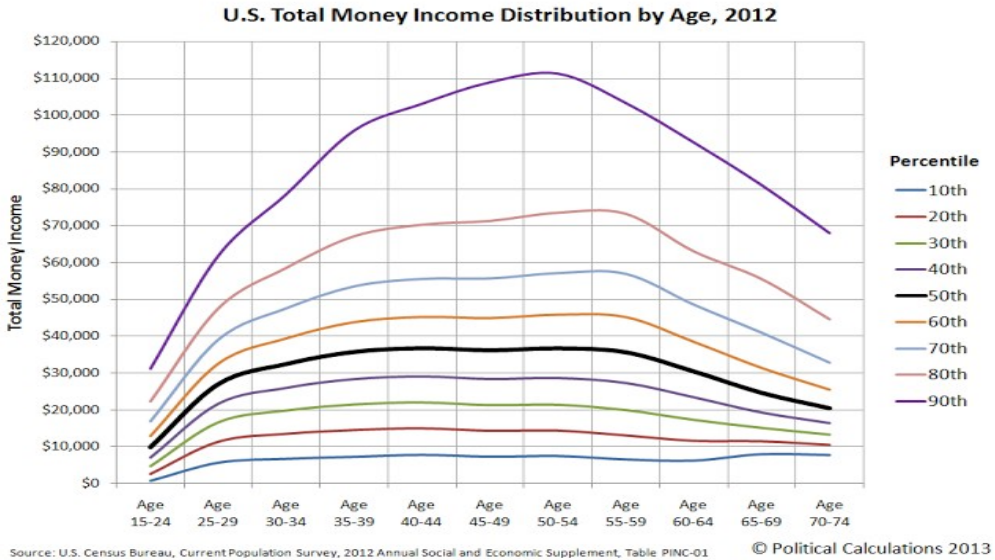
- Throughout lectures – hands-on:
  - Gradient descent
  - Intro to Tensorflow
  - Links to Deep Equilibrium Nets
  - Links to Deep Structural Estimation

DYNAMIC STOCHASTIC ECONOMIC MODELS

- **Heterogeneity** a crucial ingredient in contemporary models:
  - to study e.g. cross-sectional consumption response to aggregate shocks.
  - to model, e.g., social security.

- Example OLG models:
  - → How many age groups?
  - → borrowing constraints?
  - → aggregate shocks?
  - → idiosyncratic shocks?
  - → liquid / illiquid assets**?

→ Models: heterogeneous & high-dimensional



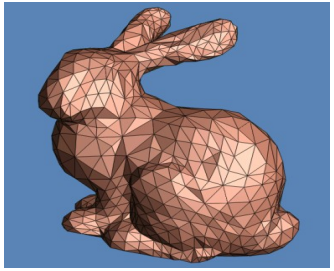
**see, e.g., Kaplan et al. (2018), Wong (2018),...

DYNAMIC STOCHASTIC ECONOMIC MODELS

e.g. Judd (1998), Ljungquist & Sargent (2004),...

$$\mathbb{E}\left[E\left(\mathbf{x}_t, \mathbf{x}_{t+1}, p\left(\mathbf{x}_t\right), p\left(\mathbf{x}_{t+1}\right)\right) | \mathbf{x}_t, p\left(\mathbf{x}_t\right)\right] = 0$$

$$\mathbf{x}_{t+1} \sim \mathcal{P}\left(\cdot | \mathbf{x}_t, p\left(\mathbf{x}_t\right)\right)$$

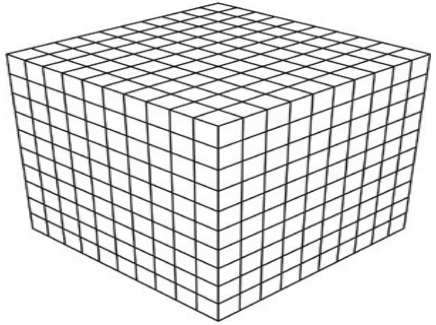


x: point in state space; describes your system.

State-space potentially irregularly-shaped and high-dimensional.

p: time-invariant policy function.

“old solution”:
high-dimensional functions on which we interpolate.



→ N^d points in ordinary discretization schemes.

→ *“Curse of dimensionality”.*

→ Need to solve many non-linear systems of equations by invoking a solver.

WHAT IS HIGH-DIMENSIONAL?

#State Variables (Dimensions)	#Points	Time-to-solution
1	10	10 sec
2	100	~ 1.6 min
3	1,000	~ 16 min
4	10,000	~ 2.7 hours
5	100,000	~ 1.1 days
6	1,000,000	~ 1.6 weeks
…	…	…
20	1e20	3 trillion years (240x age of the universe)

Dimension reduction
Exploit symmetries, e.g., via the active subspace method

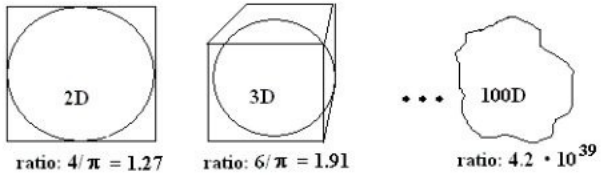
Deal with #Points

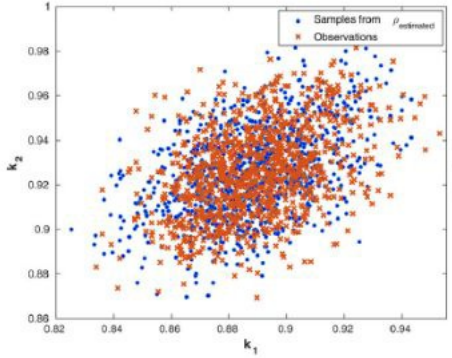
High-performance computing
Reduces time to solution, but not the problem size

VOLUMES IN HIGH DIMENSIONS

- Consider a cube of unit lengths containing a sphere of unit radius in higher dimensions.

- For large dimensions: ratio Volume(Sphere)/Volume(Cube) → 0



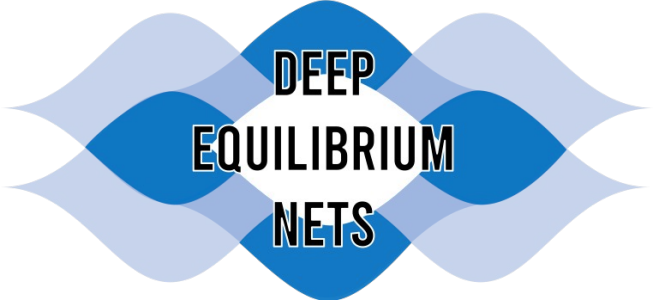


Scheidegger & Bilionis (2019)

ABSTRACT PROBLEM FORMULATION

- Contemporary dynamic models: heterogeneous & high-dimensional
- Want to solve dynamic stochastic models with high-dimensional state spaces
  - Have to approximate and interpolate high-dimensional functions on irregular-shaped geometries
  - Problem: curse of dimensionality
- Want to alleviate the curse of dimensionality
- Want locality of approximation scheme
- Speed-up → access hybrid HPC systems
- Confront model to data (structural estimation – fast model eval. needed)

DEEP EQUILIBRIUM NETS



- Joint work with M. Azinovic (UZH), L. Gaegauf (UZH)
  https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3393482

- Code examples: https://github.com/sischei/DeepEquilibriumNets

- We introduce the concept of **Deep Equilibrium Nets**.
  - **Neural networks** that directly approximate all equilibrium functions in discrete-time dynamic stochastic economic models.
  - They are trained in an **unsupervised** fashion to satisfy all equilibrium conditions along simulated paths of the economy.

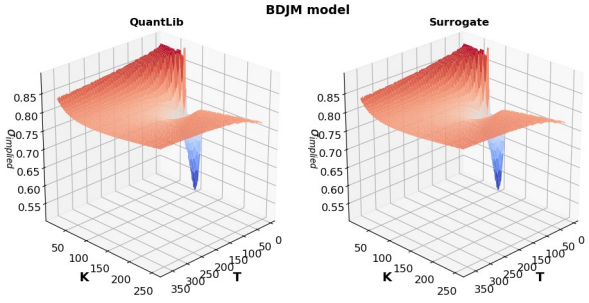
- Neural network approximates the equilibrium functions directly

  → neither sets of non-linear equations nor optimization problems need to be solved in order to simulate the economy.

  → **Training data can be generated at virtually zero cost.**

  → **Generic, scalable & flexible global solution method.**

DEEP STRUCTURAL ESTIMATION



- DSE: joint work with A. Didisheim, H. Chen (MIT)
  https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3782722
- Code examples: https://github.com/DeepStructuralEstimation/OptionPricing
- We solve models as functions of their economic states and parameters (=pseudo-states).
- This high-dimensional solution is numerically approximated with Deep Neural Networks (called a "surrogate").
- Computationally cheap to evaluate, i.e., to confront the surrogate to data.

THE RISE OF NEURAL NETWORKS

TOM SIMONITE BUSINESS 01.25.15 01:05 PM

DEEPMIND BEATS PROS AT STARCRAFT IN ANOTHER TRIUMPH FOR BOTS

A pro gamer and an AI bot duke it out in the strategy game StarCraft, which has becor bed for research on artificial intelligence. STARCRAFT

IN LONDON LAST month, a team from Alphabet's UK-ba artificial intelligence research unit DeepMind quietly l new marker in the contest between humans and compu On Thursday it revealed the achievement in a three-ho YouTube stream, in which aliens and robots fought to t death.

TOM SIMONITE BUSINESS 10.18.17 01:00 PM

THIS MORE POWERFUL VERSION OF ALPHAGO LEARNS ON ITS OWN

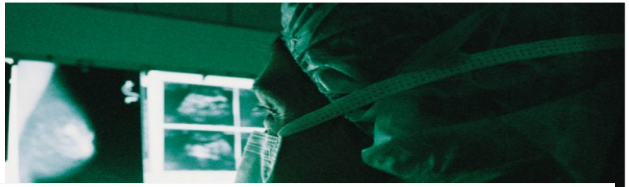
NOAH SHELDON FOR WIRED

AT ONE POINT during his historic defeat to AlphaGo last year, world champion Go play abruptly left the room. The bot had played confounded established theories of the bo moment that came to epitomize the myste AlphaGo.

NEWS BIOLOGY 21 DECEMBER 2017

AI beats docs in cancer spotting

A new study provides a fresh example of machine learning as an important diagnostic tool. Paul Biegler reports.



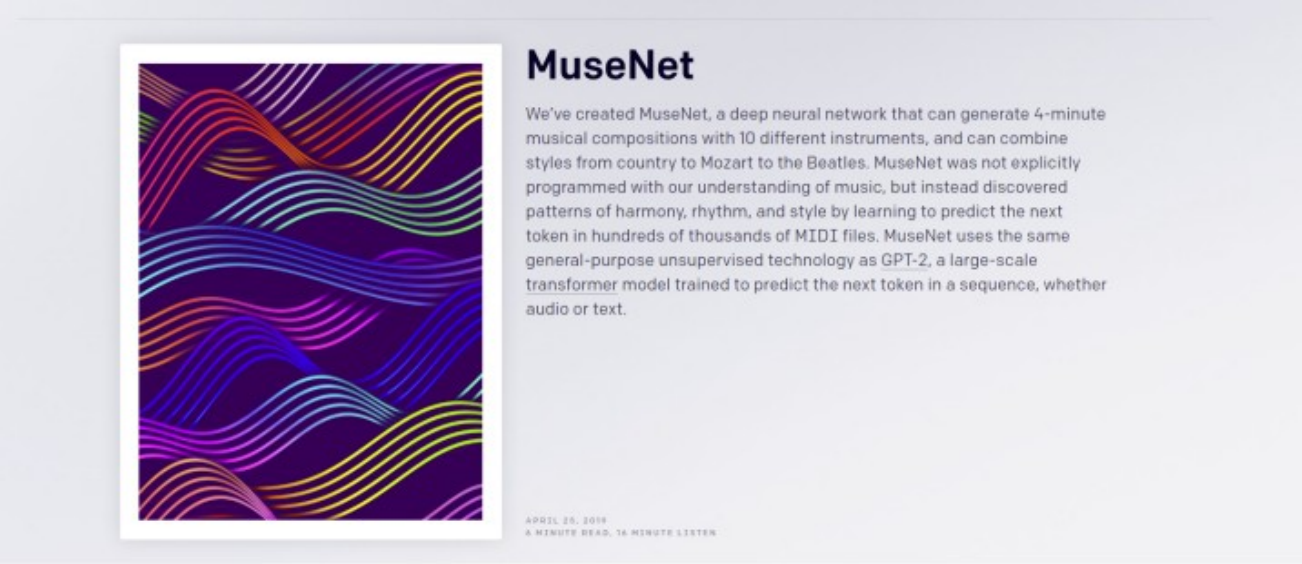
Deep Learning Software Speeds Up Drug Discovery

Wed, 01/16/2019 - 8:00am 1 Comment by Kenny Walter , Science Reporter - @RandDMagazine

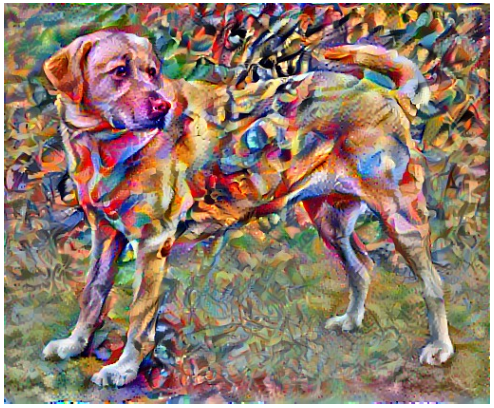
The long, arduous process of narrowing down millions of chemical compounds to just a select few that can be further developed into mature drugs, may soon be shortened, thanks to new artificial intelligence (AI) software.

MUSIC GENERATED BY AI

- https://openai.com/blog/musenet/



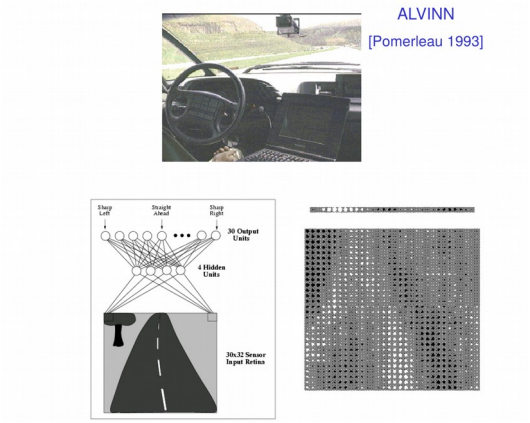
STYLE TRANSFER



Figs. from https://www.tensorflow.org/tutorials/generative/style_transfer

SELF-DRIVING CARS

- Carnegie Mellon University – 1990ies
  - ALVINN: Autonomous Land Vehicle In a Neural Network



- Today (e.g., Waymo)
  - https://www.youtube.com/watch?v=LSX3qdy0dFg

APP: COLORING OLD MOVIES

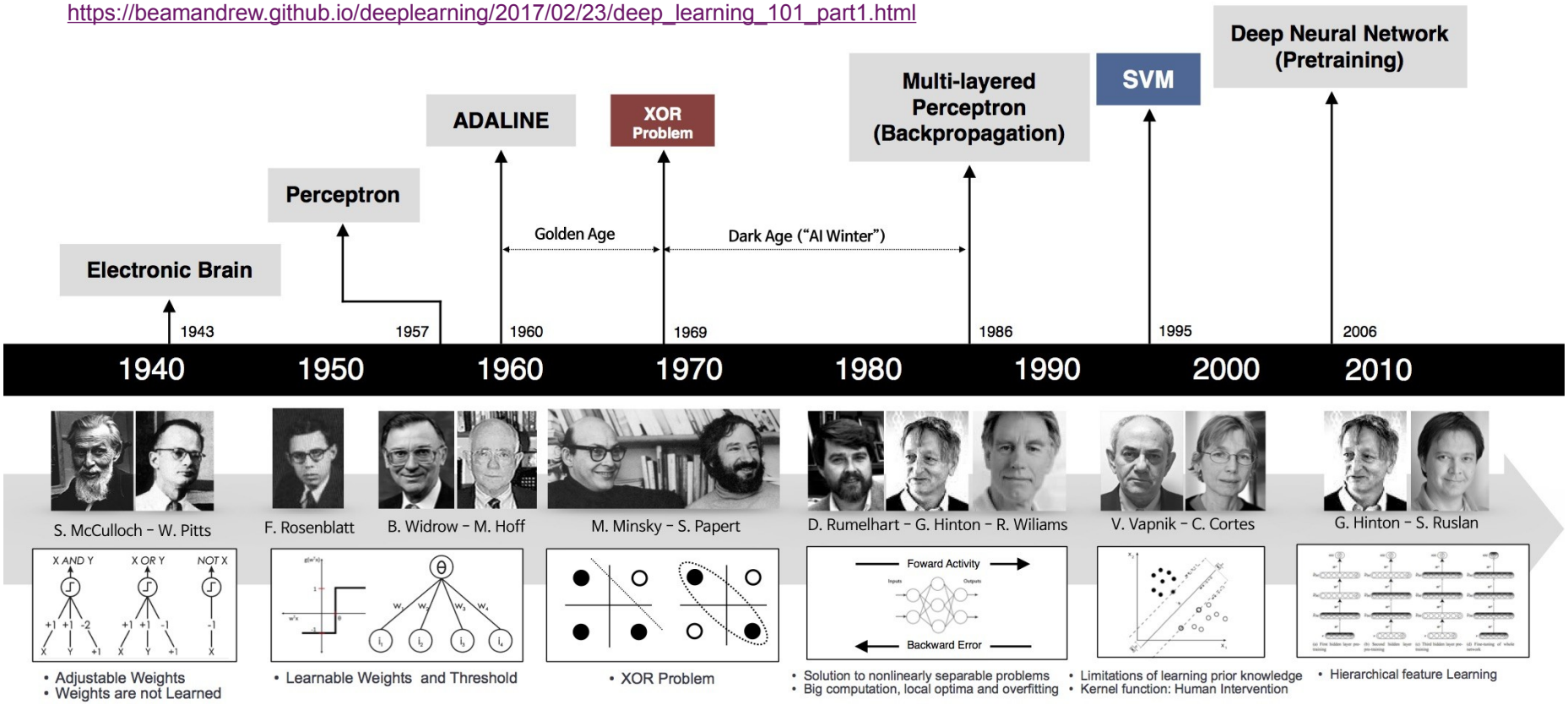
- https://deepsense.ai/ai-movie-restoration-scarlett-ohara-hd/

TWO-LEGGED ROBOTS

- https://www.youtube.com/watch?v=hSjKoEva5bg&feature=youtu.be

A TIMELINE OF DEEP LEARNING

https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html



*J. Schmidhuber clearly missing on this slide

WHY NOW?

- Neural Networks date back decades, so why the resurgence?

  (Stochastic Gradient Descent: 1952, Perceptron: 1958, Back-propagation: 1986, Deep Convolutional NN: 1995)

  - **Big Data**
    - Large Datasets
    - Easier Collection and Storage
  - **Hardware**
    - GPUs, TPUs,…
  - **Software**
    - Improved Techniques
    - Toolboxes

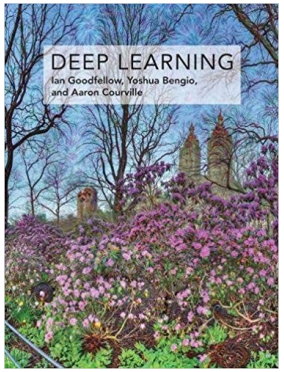
TensorFlow PyTorch

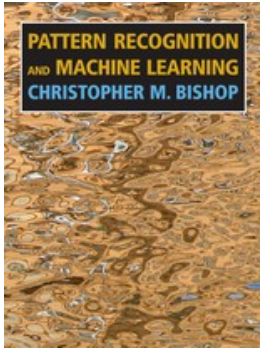
EXPECTATIONS MANAGEMENT

- This is a super-faced paced lecture.
- We will have only time to cover the basics.
- After those lectures, you will need to practice, practice, practice to become a master.

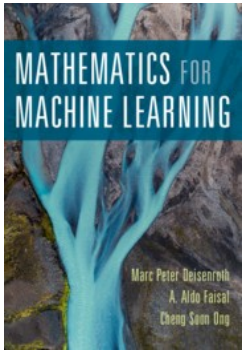
SOME USEFUL MATERIALS

Deep Learning
Ian Goodfellow and Yoshua Bengio and Aaron Courville
MIT Press 2016
http://www.deeplearningbook.org/





Pattern Recognition and Machine Learning
C. M Bishop, Springer 2006
(pdf freely available)

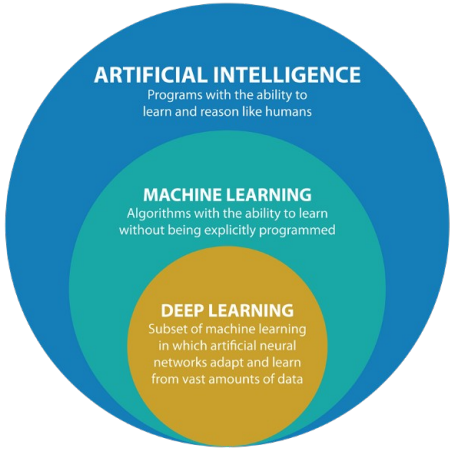


Mathematics for Machine Learning
Deisenroth, A. Aldo Faisal, and Cheng Soon Ong.
Cambridge University Press 2020

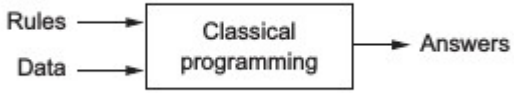
→ ***There is a great community out there (use your browser and Google around...)***

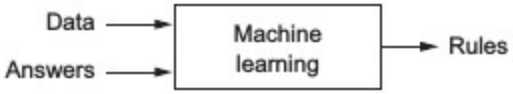
RECAP ON MACHINE LEARNING

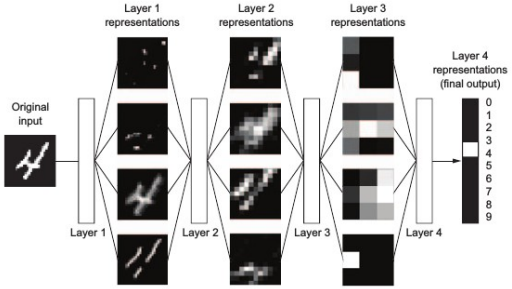
SOME TERMINOLOGY



- **Artificial intelligence (AI)**
  - Can computers be made to “think”—a question whose ramifications we’re still exploring today.
  - A concise definition of the field would be as follows: the effort to automate intellectual tasks normally performed by humans.
- **Machine learning (e.g., supervised ML)**







- **Deep Learning as a particular example of an ML technique**

Figs. from F. Chollet (2018) – ML with Python

TYPES OF MACHINE LEARNING

- **Supervised Learning**
  - assume that training data is available from which they can learn to predict a target feature based on other features (e.g., monthly rent based on area).

    - **Classification**
    - **Regression**

- **Unsupervised Learning**
  - take a given data-set and aim at gaining insights by identifying patterns, e.g., by grouping similar data points.

- **Reinforcement Learning**

SUPERVISED REGRESSION

- Regression aims at predicting a numerical target feature based on one or multiple other (numerical) features.

- Example: Price of a used car.
  - x : car attributes
  - y : price
  - $y = h(x \mid \theta)$
  - h ( ): model
  - θ : parameters

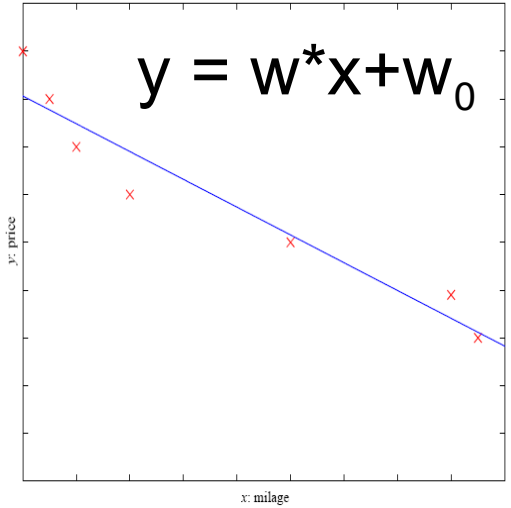


Fig. from Alpaydin (2014)

SUPERVISED CLASSIFICATION

Example 1: Spam Classification

- Decide which emails are Spam and which are not.
- Goal: Use emails seen so far to produce a good prediction
- rule for **future** data.

Example 2: Credit Scoring

- Differentiating between low-risk and high-risk customers from their income and savings.

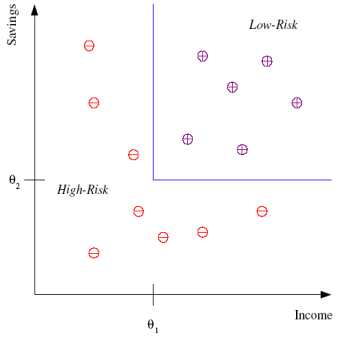
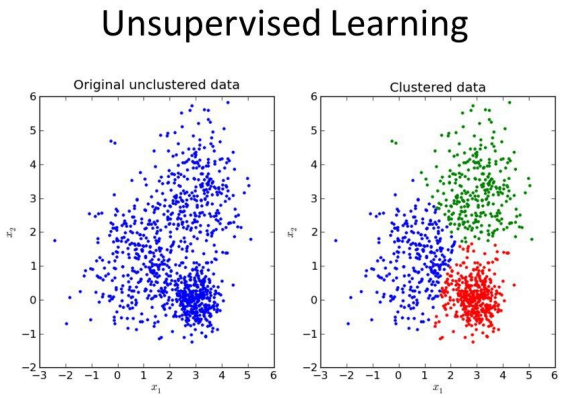


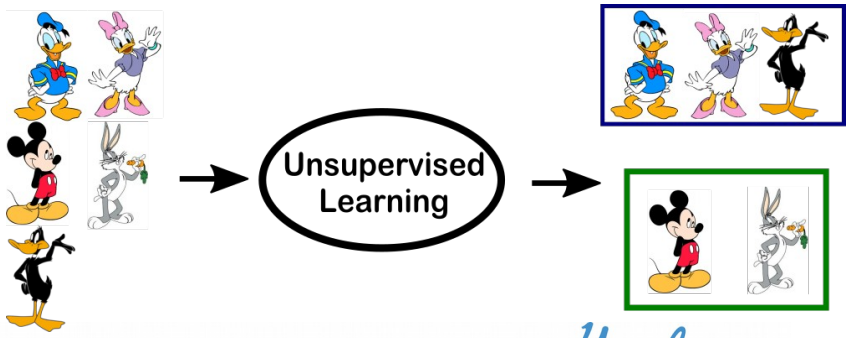
Fig. from Alpaydin (2014)

- Discriminant: IF income > θ_2 AND savings > θ_2
  THEN low-risk ELSE high-risk

UNSUPERVISED ML

- No output
- Clustering: Grouping similar instances
- Example applications:
  - Customer segmentation
  - Image compression
  - Bio-informatics: Learning motifs
  - ...





REINFORCEMENT LEARNING

- Learning a policy: A sequence of outputs

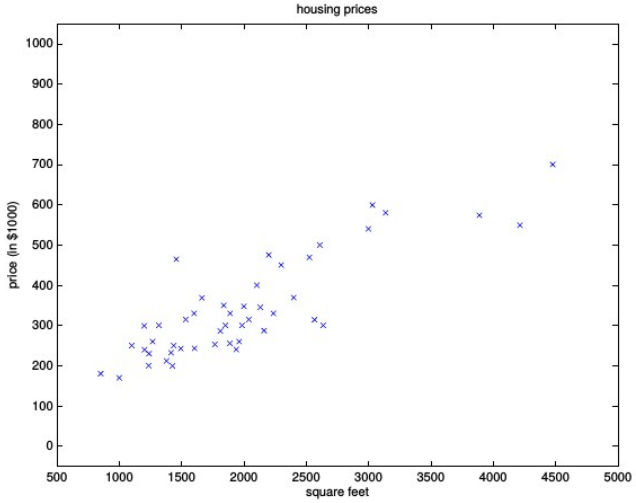
- No supervised output but delayed reward
  - Game playing
  - Robot in a maze
  - …
- See, e.g., https://www.youtube.com/watch?v=V1eYniJ0Rnk&vl=en

BUILDING AN ML ALGORITHM

- Optimize a performance criterion using example data or past experience.
- Role of statistics: Inference from a sample.
- Role of computer science: Efficient algorithms to
  - Solve the optimization problem.
  - Representing and evaluating the model for inference.

BUILDING AN ML ALGORITHM (II)

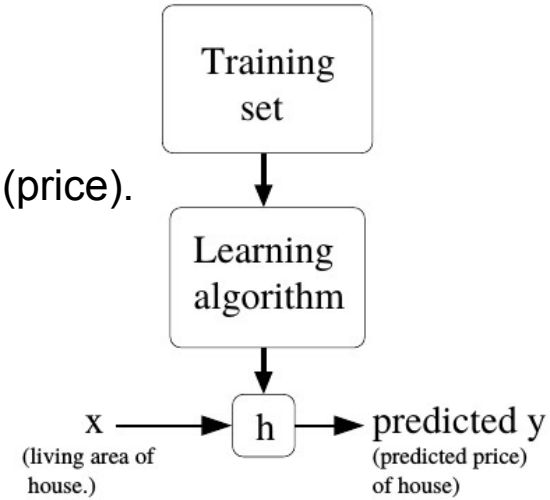
Living area ($feet^2$)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮



- Given data like this, how can we learn to predict the prices of other houses as a function of the size of their living areas?

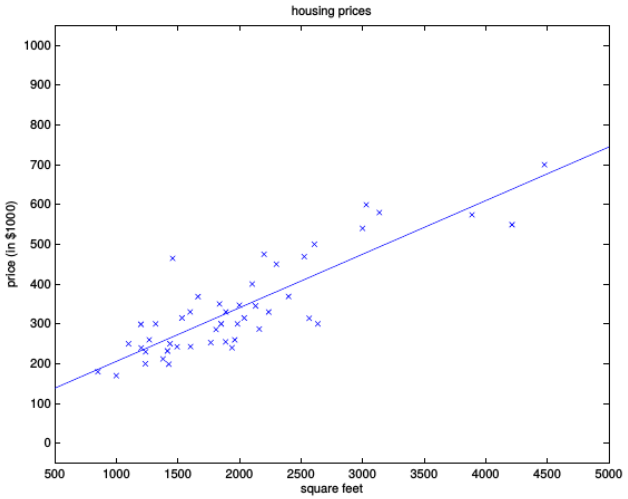
BUILDING AN ML ALGORITHM (III)

- **x(i): "input" variables** (living area in this example), also called **input features**
- **y(i): "output" / target variable** that we are trying to predict (price).
- **Training example**: a pair (x(i) , y(i)).
- **Training set**: a list of m training examples {(x(i), y (i)); i = 1, . . . , m}
  - To perform supervised learning, we must decide how we're going to represent **functions/hypotheses *h*** in a computer.



BUILDING AN ML ALGORITHM (IV)

- Model / Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x_1$
  - θ_i's: parameters



- Cost Function:

$$\mathrm{J}(\theta) = \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2$$

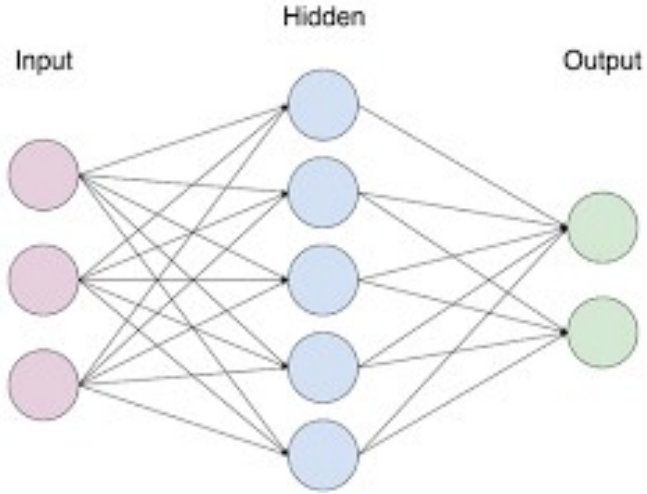
  - Minimize J(θ) in order to obtain the coefficients θ.

BUILDING AN ML ALGORITHM (V)

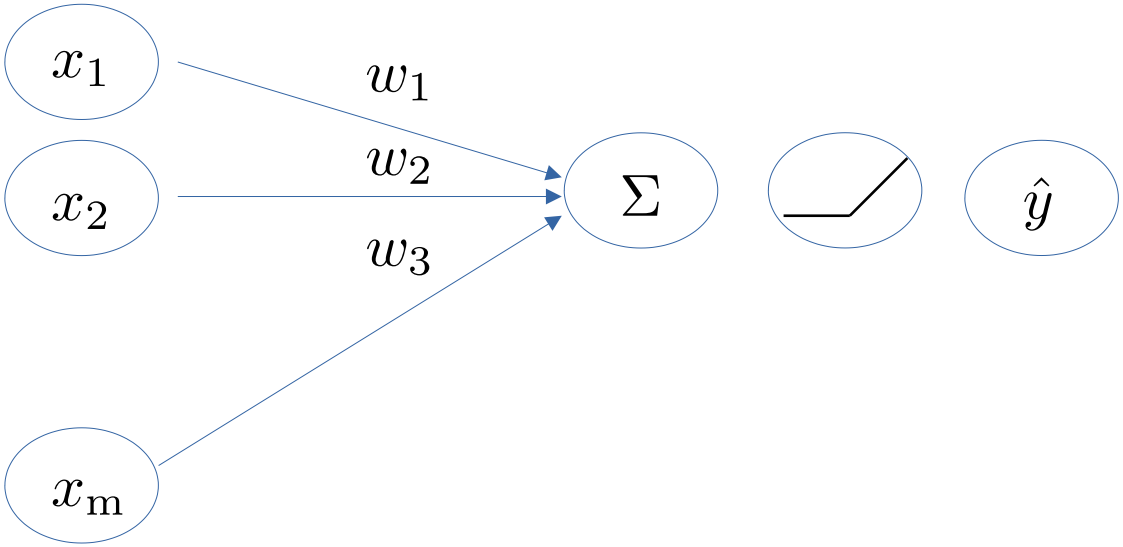
- In General: Machine learning in 3 steps:
  - Choose a **model** $h(x|\theta)$.
  - Define a **cost function** $J(\theta|x)$.
  - **Optimization procedure** to find θ^* that minimizes $J(\theta)$.

- Computationally, we need:
  - data, linear algebra, statistics tools, and optimization routines.

ARTIFICIAL NEURAL NETWORKS



THE PERCEPTRON: FORWARD PROPAGATION



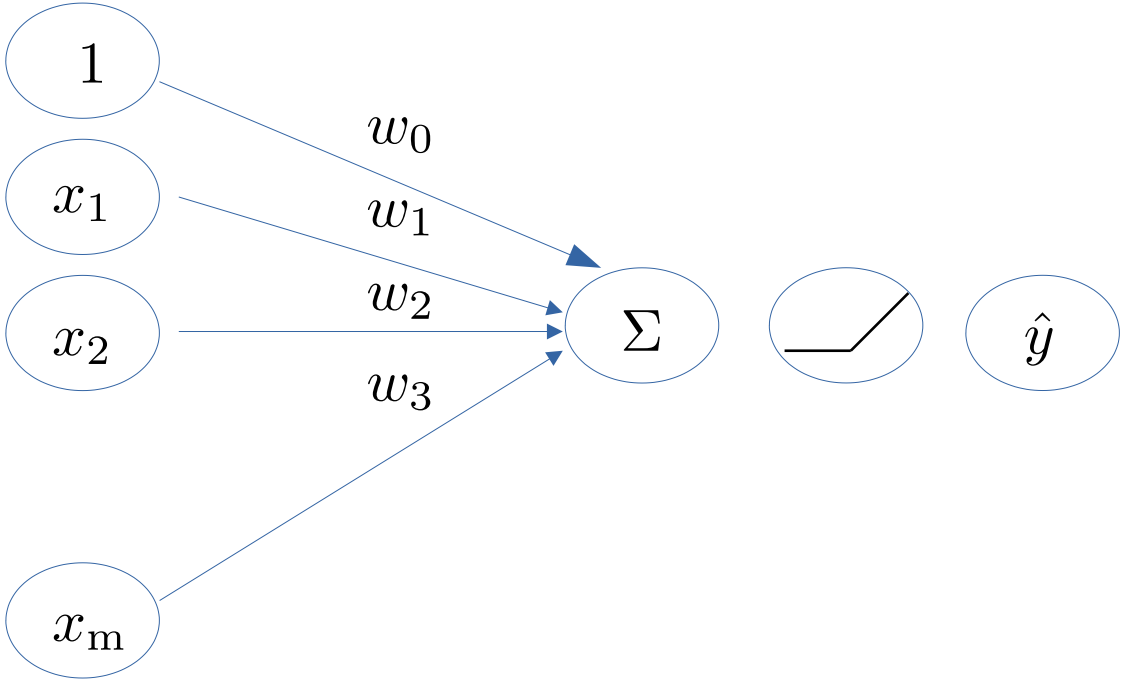
Output Linear combination of inputs

$$\hat{y} = g\left(\sum_{i=1}^{m} x_i w_i\right)$$

Non-linear activation function

Inputs Weights Sum Non-Linearity Output

THE PERCEPTRON: FORWARD PROPAGATION



Output Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

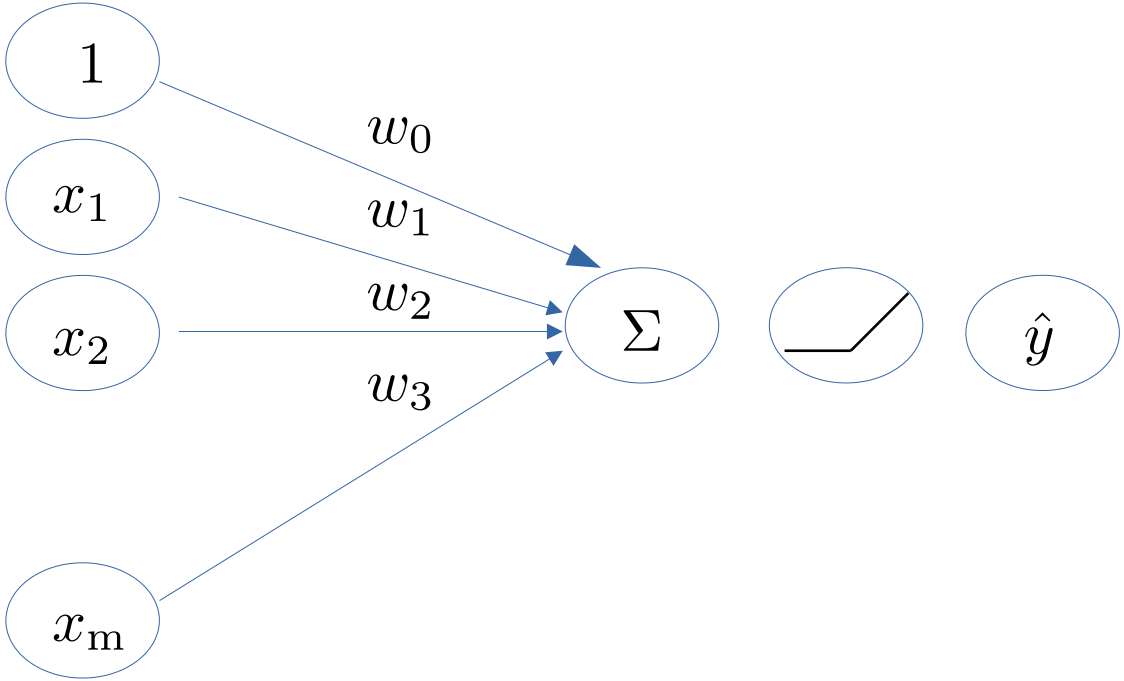
Bias

Non-linear activation function

Inputs Weights Sum Non-Linearity Output

→ **Bias term allows you to shift your activation function to the left or the right**

THE PERCEPTRON: FORWARD PROPAGATION



$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

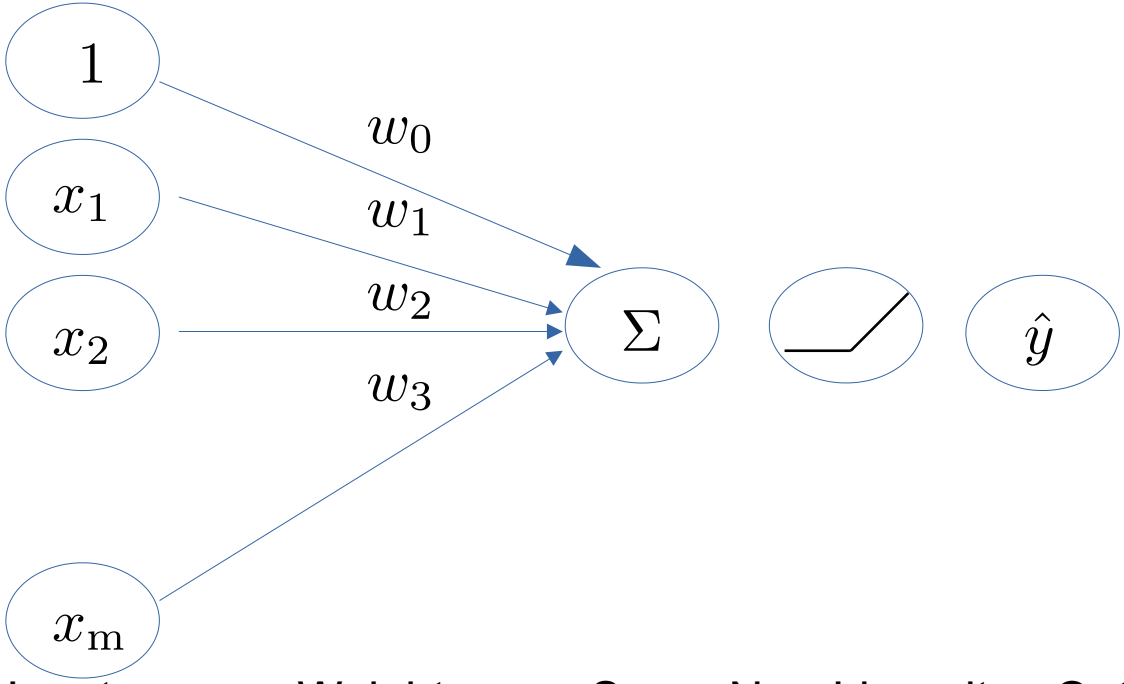
$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

$$\begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Inputs Weights Sum Non-Linearity Output

→ **Bias term allows you to shift your activation function to the left or the right**

THE PERCEPTRON: FORWARD PROPAGATION

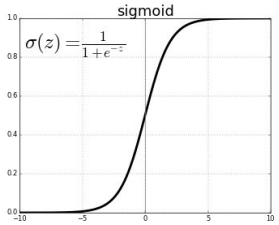


Inputs Weights Sum Non-Linearity Output

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

Activation Functions
e.g. sigmoid function

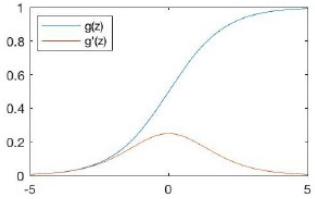
$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$



→ **Bias term allows you to shift your activation function to the left or the right**

FEW ACTIVATION FUNCTIONS

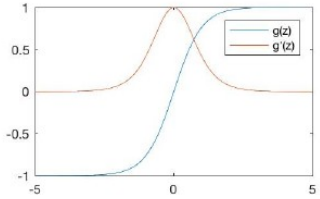
Sigmoid Function



$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

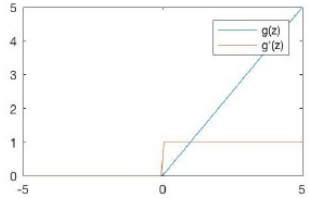
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

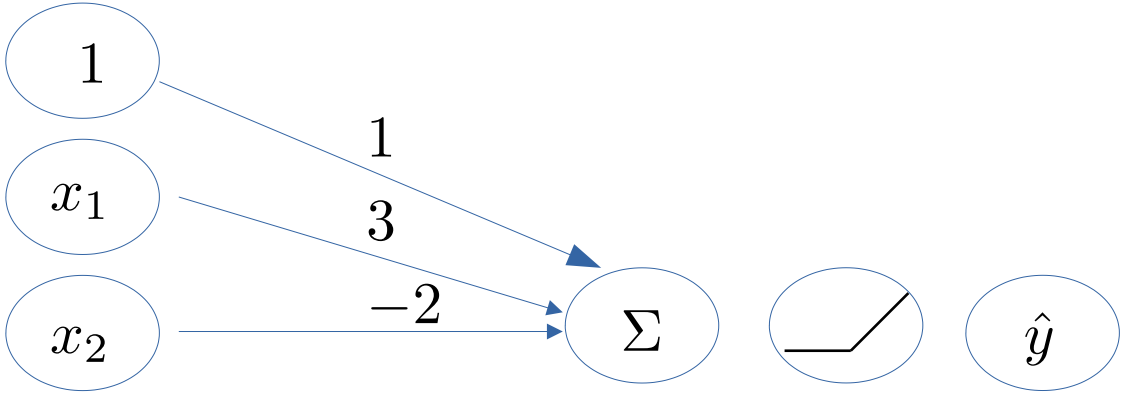


$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Needs to be differentiable for gradient-based learning (later)
  - Very useful in practice.
  - Sigmoid function, e.g., useful for classification (Probability).

PERCEPTRON – AN EXAMPLE



We have: $w_0 = 1$ and $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

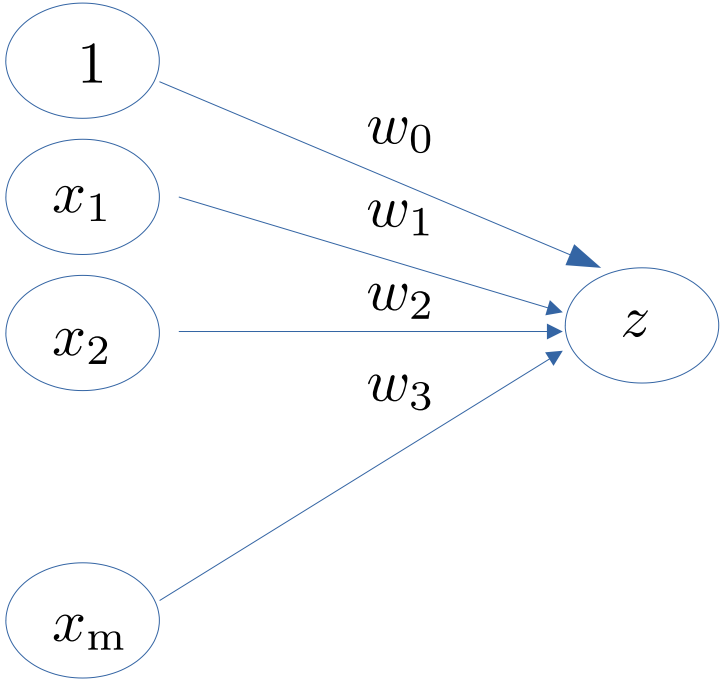
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$

This is just a line in $2D$

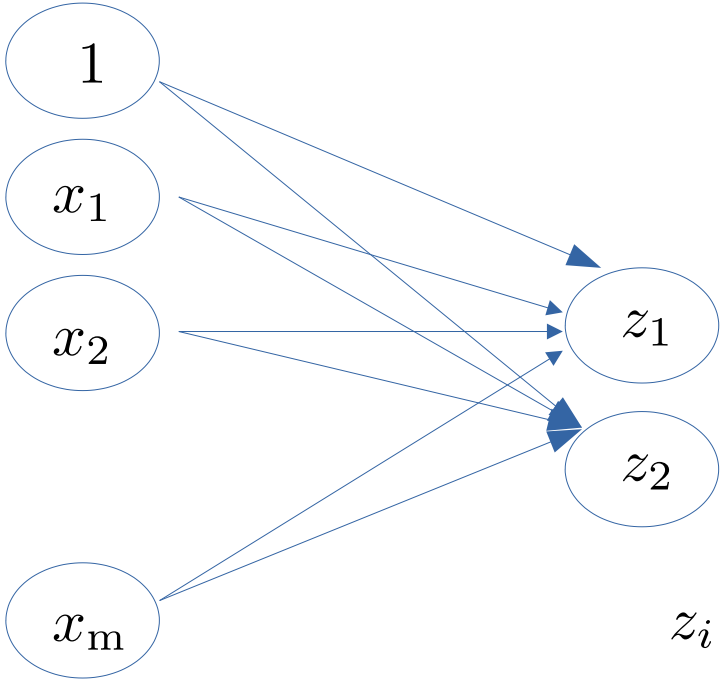
Imagine we have a trained network with weights given.
→ how do we compute the output?

A PERCEPTRON – SIMPLIFIED



$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$

BUILDING A NN WITH PERCEPTRONS: A MULTI-OUTPUT PERCEPTRON

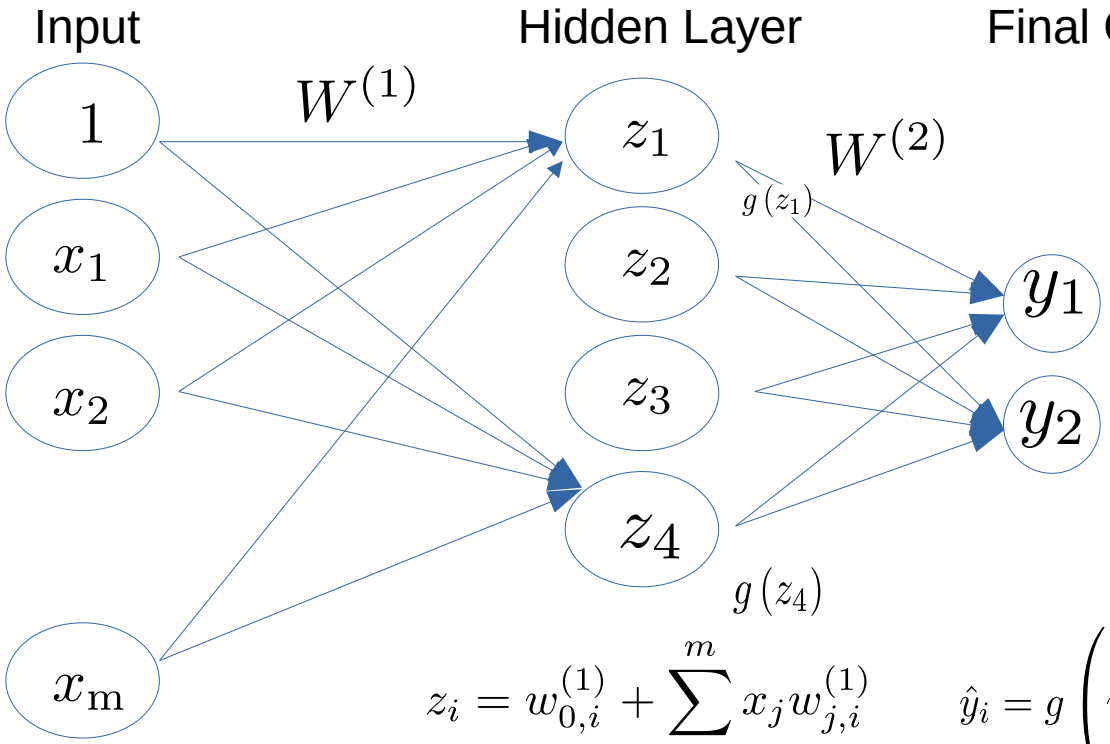


$$y_1 = g(z_1)$$

$$y_2 = g(z_2)$$

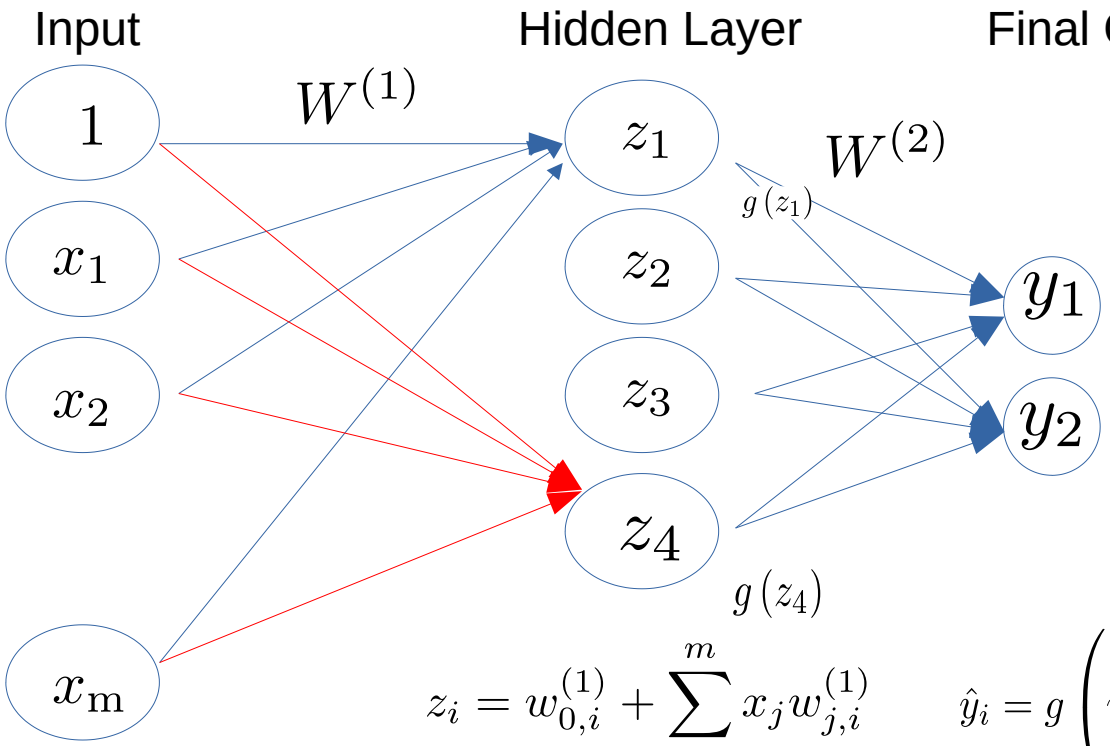
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$

SINGLE LAYER NN



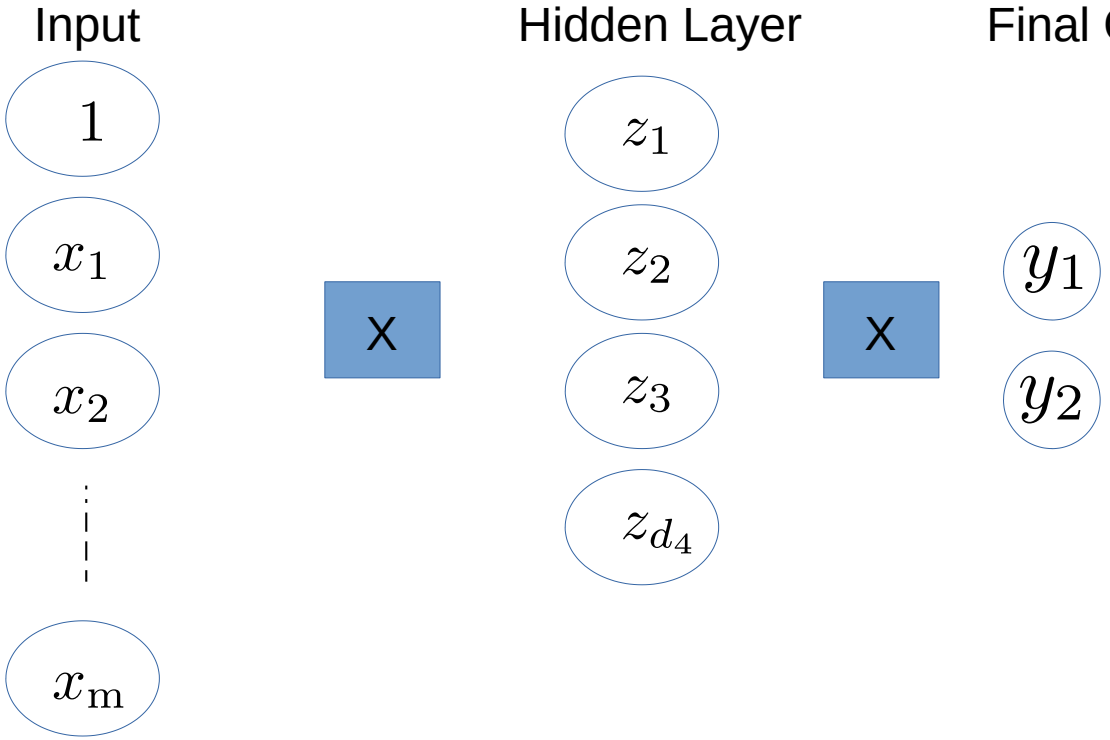
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

SINGLE LAYER NN

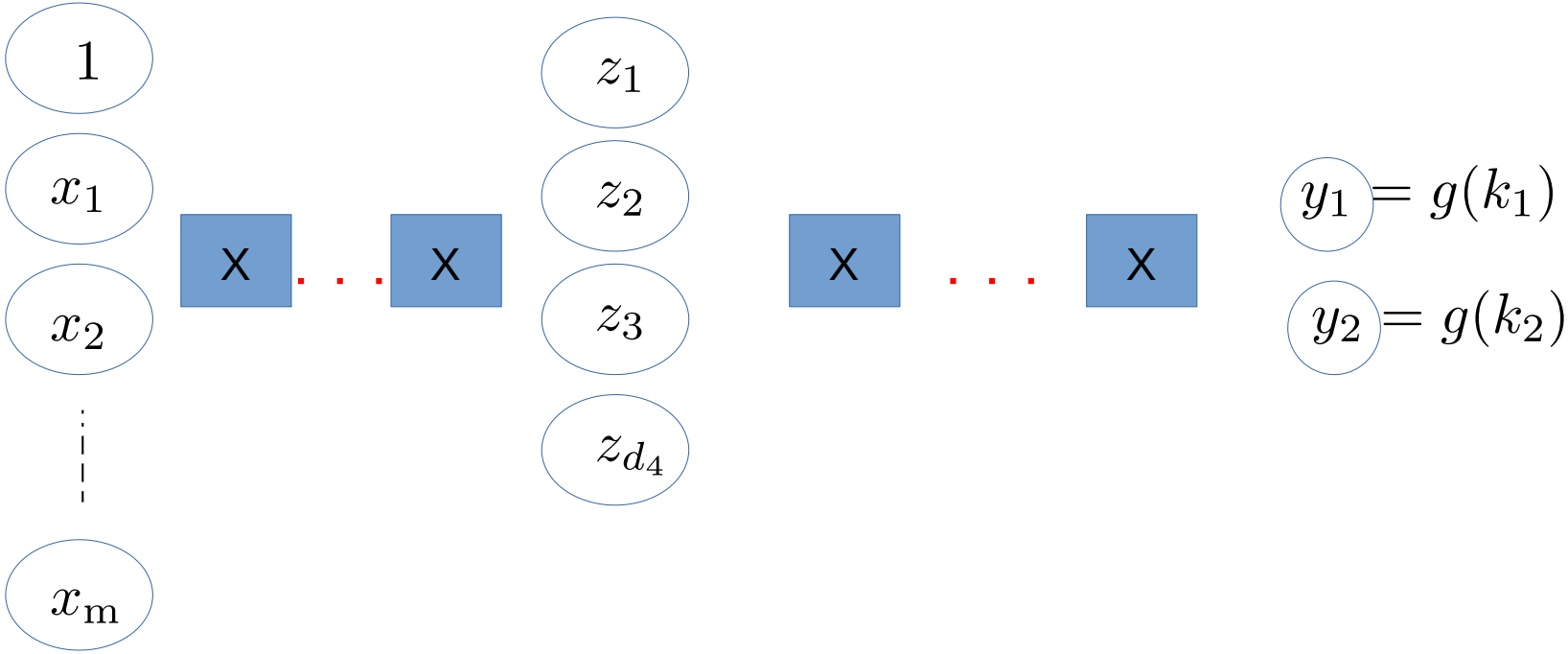


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

SINGLE LAYER NN



FULLY CONNECTED DNN



EXPRESSIVENESS OF ANN

- **Boolean functions:**
  - Every Boolean function can be represented by a network with a single hidden layer.
  - Might require exponential (in number of inputs) hidden units.

- **Continuous functions:**
  - Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989].

- Deep NN are in practice superior to other ML methods in presence of large data sets.

- Little known about convergence rates/breaking the curse of dimensionality in context with particular architectures (Dou & Montanelli (2019) for special cases).

- In practice verification, valildation, and uncertainty quantification important (VVUQ) to demonstacte the “correctness” of the numerical results!

UNIVERSAL FCT. APPROXIMATOR

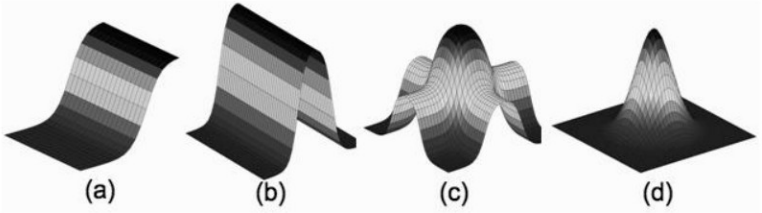


FIGURE 4.9 The learning of the MLP can be shown as the output of a single sigmoidal neuron (a), which can be added to others, including reversed ones, to get a hill shape (b). Adding another hill at 90° produces a bump (c) , which can be sharpened to any extent we want (d), with the bumps added together in the output layer. Thus the MLP learns a local representation of individual inputs.

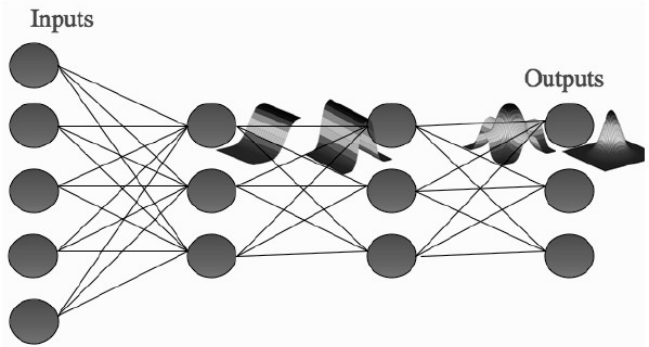


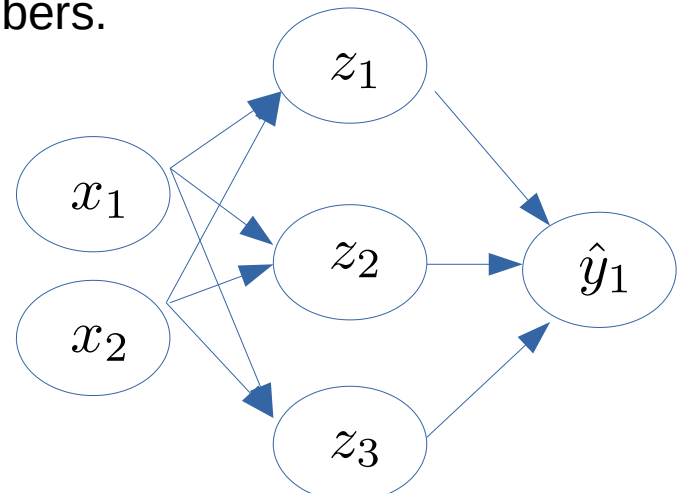
FIGURE 4.10 Schematic of the effective learning shape at each stage of the MLP.

Figs. from Marsland (2014)

MEAN SQUARED ERROR (MSE)

Mean squared error can be used with regression models that output continuous real numbers.

$$\begin{bmatrix} 80.000, 200.000 \\ 120.000, 400.000 \\ 10.000, 12.000 \\ ..., ... \end{bmatrix}$$



f(x)

$$\begin{bmatrix} 450.000 \\ 250.000 \\ 190.000 \\ ..., ... \end{bmatrix}$$

y

$$\begin{bmatrix} 470.000 \\ 220.000 \\ 250.000 \\ ..., ... \end{bmatrix}$$

Loan requested

Loan required

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - f\left(x^{(i)}; W \right) \right)^2$$

Actual Predicted

NETWORK TRAINING

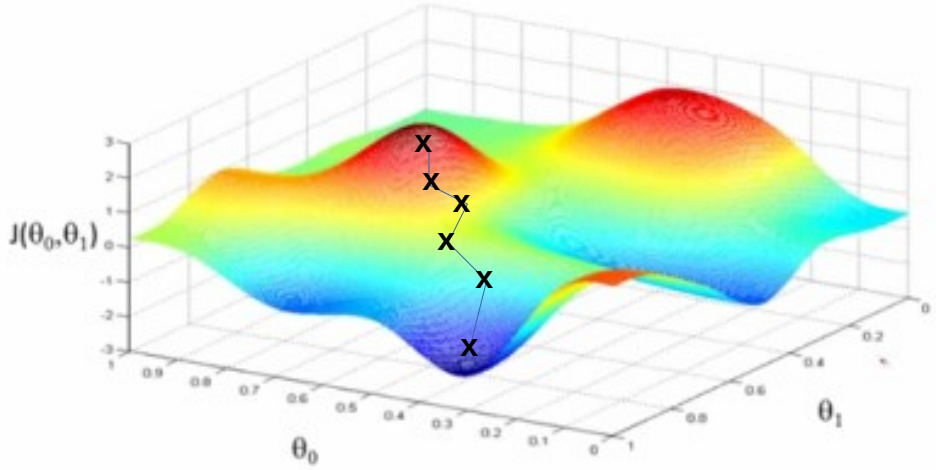
We want to find the network weights that achieve the lowest loss!

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{W}{\operatorname{argmin}} J(\boldsymbol{W})$$

GRADIENT DESCENT IN WEIGHT SPACE

- $W^* = \underset{W}{\text{argmin}} J(W)$
- Randomly pick an initial(w_0, w_1)
- Compute gradient
- Take small steps in the opposite direction of gradient.
- Repeat until convergence



DESCENT METHODS IN GENERAL

- The typical strategy for optimization problems of this sort is a descent method:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \Delta\boldsymbol{w}^{(\tau)}$$

- These come in many flavors
  - Gradient descent $\nabla J\left(\boldsymbol{w}^{(\tau)}\right)$
  - Stochastic gradient descent $\nabla J_n\left(\boldsymbol{w}^{(\tau)}\right)$
  - Newton-Raphson (second order) ∇^2

- All of these can be used here, stochastic gradient descent is particularly effective
  - Redundancy in training data, escaping local minima.

GRADIENT DESCENT ALGORITHM

Algorithm

I. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

2. Loop until convergence:

3. Compute gradient, $\frac{\partial J(W)}{\partial W}$ ← Can be computationally expensive

4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial W}$

5. Return weights

GRADIENT DESCENT ALGORITHM

Algorithm

I. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

2. Loop until convergence:

3. Compute gradient, $\frac{\partial J(W)}{\partial W}$

4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial W}$

5. Return weights

All that matters to train a NN

Learning rate

STOCHASTIC GRADIENT DESCENT

Algorithm

I. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

2. Loop until convergence:

3. Pick single data point i

4. Compute gradient, $\frac{\partial J_i(W)}{\partial W}$

5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

6. Return weights

Can be noisy

STOCHASTIC GRADIENT DESCENT

Algorithm

I. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

2. Loop until convergence:

3. Pick batch of B data points

4. Compute gradient, $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$

5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

6. Return weights

MINI-BATCHES WHILE TRAINING

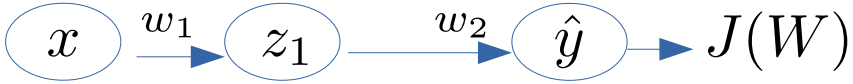
- More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates

- Mini-batches lead to fast training!
  - Can parallelize computation + achieve significant speed increases on GPU's

- Note: a complete pass over all the patterns in the training set is called an **epoch**.

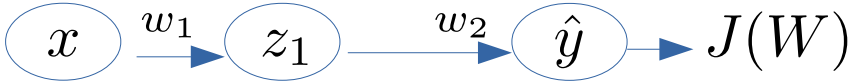
COMPUTING GRADIENTS: ERROR BACKPROPAGATION

- How does a small change in one weight (e.g., w_2) affect the final loss J(W)?



COMPUTING GRADIENTS: ERROR BACKPROPAGATION

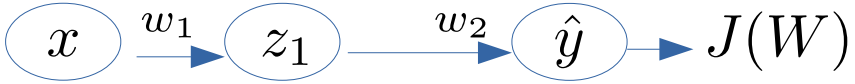
- How does a small change in one weight (e.g., w_2) affect the final loss J(W)?
- Chain rule



$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

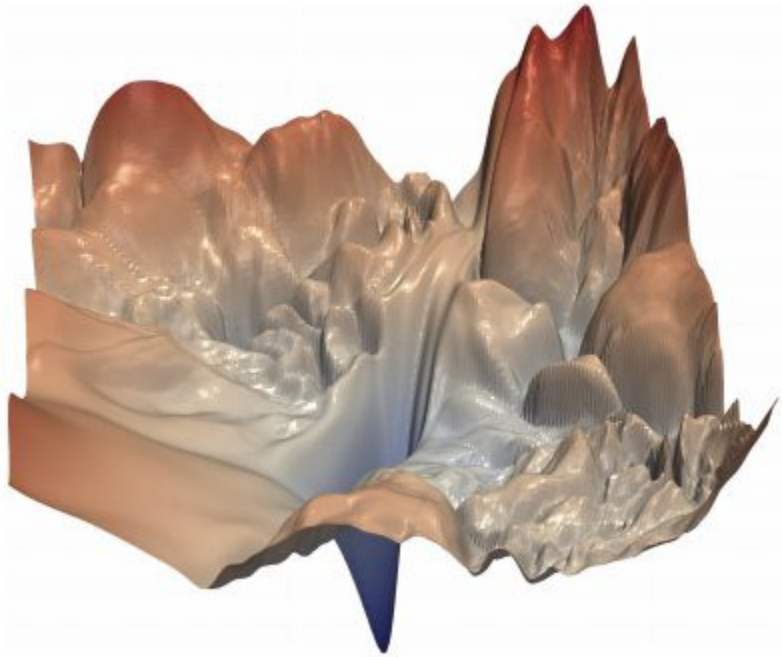
COMPUTING GRADIENTS: ERROR BACKPROPAGATION

- How does a small change in one weight (e.g., w_2) affect the final loss J(W)?
- Chain rule
- **Repeat this for every weight in the network using gradients from later layers**



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1} \longrightarrow \frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

TRAINING NEURAL NETWORKS



See https://papers.nips.cc/paper/7875-visualizing-the-loss-landscape-of-neural-nets.pdf

Bonus slide

LOSS FUNCTION: CAN BE DIFFICULT TO OPTIMIZE

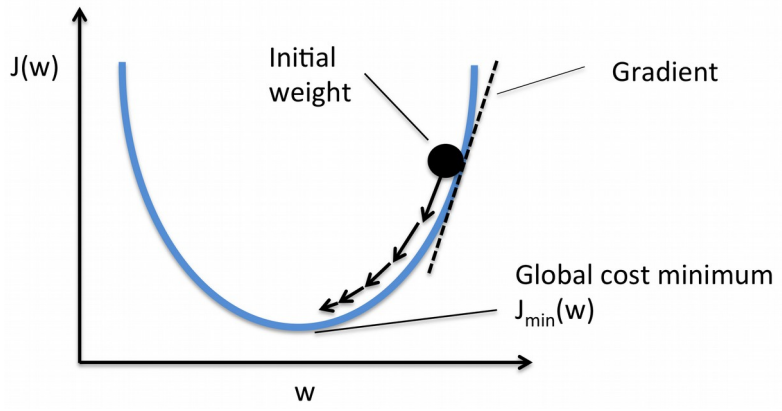
- Remember:
  - Optimization through gradient descent: $$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$
  - How can we set the learning rate?

MISSPECIFIED REWARD FUNCTION

- https://www.youtube.com/watch?v=tlOIHko8ySg
- https://openai.com/blog/faulty-reward-functions/

SETTING THE LEARNING RATE

- Small learning rate converges slowly and gets stuck in false local minima
  - Design an adaptive learning rate that “adapts” to the landscape.



FEW VARIANTS OF SGD

Method	Formula
Learning Rate	$w^{(t+1)} = w^{(t)} - \eta \cdot \nabla \ell(w^{(t)}, z) \quad = w^{(t)} - \eta \cdot \nabla w^{(t)}$
Adaptive Learning Rate	$w^{(t+1)} = w^{(t)} - \eta_t \cdot \nabla w^{(t)}$
Momentum [Qian 1999]	$w^{(t+1)} = w^{(t)} + \mu \cdot (w^{(t)} - w^{(t-1)}) - \eta \cdot \nabla w^{(t)}$
Nesterov Momentum [Nesterov 1983]	$w^{(t+1)} = w^{(t)} + v_t; \quad v_{t+1} = \mu \cdot v_t - \eta \cdot \nabla \ell(w^{(t)} - \mu \cdot v_t, z)$
AdaGrad [Duchi et al. 2011]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A_{i,t}} + \varepsilon}; \quad A_{i,t} = \sum_{\tau=0}^{t} \left(\nabla w_i^{(t)} \right)^2$
RMSProp [Hinton 2012]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A'_{i,t}} + \varepsilon}; \quad A'_{i,t} = \beta \cdot A'_{t-1} + (1-\beta) \left(\nabla w_i^{(t)} \right)^2$
Adam [Kingma and Ba 2015]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot M_{i,t}^{(1)}}{\sqrt{M_{i,t}^{(2)}} + \varepsilon}; \quad M_{i,t}^{(m)} = \frac{\beta_m \cdot M_{i,t-1}^{(m)} + (1-\beta_m) \left(\nabla w_i^{(t)} \right)^m}{1-\beta_m^t}$

For more details, see https://arxiv.org/pdf/1609.04747.pdf, and also https://ruder.io/optimizing-gradient-descent/

WEIGHT INITIALIZATION

- Before the training process starts: all weights vectors must be initialized with some numbers.
- There are many initializers of which random initialization is one of the most widely known ones (e.g., with a normal distribution).
  - Specifically, one can configure the mean and the standard deviation, and once again seed the distribution to a specific (pseudo-)random number generator.
  - which distribution to use, then?
  - random initialization itself can become problematic under some conditions: you may then face the vanishing gradients and exploding gradients problems.
- What to do against these problems?
  - e.g. Xavier & He initialization (available in Keras)
  - They are different in the way how they manipulate the drawn weights to arrive at approximately 1. By consequence, they are best used with different activation functions.
  - Specifically, He initialization is developed for ReLU based activating networks and by consequence is best used on those. For others, Xavier (or Glorot) initialization generally works best.

See, e.g., Glorot & Bengio (2010) - http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf; He et al. (2015) - https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf

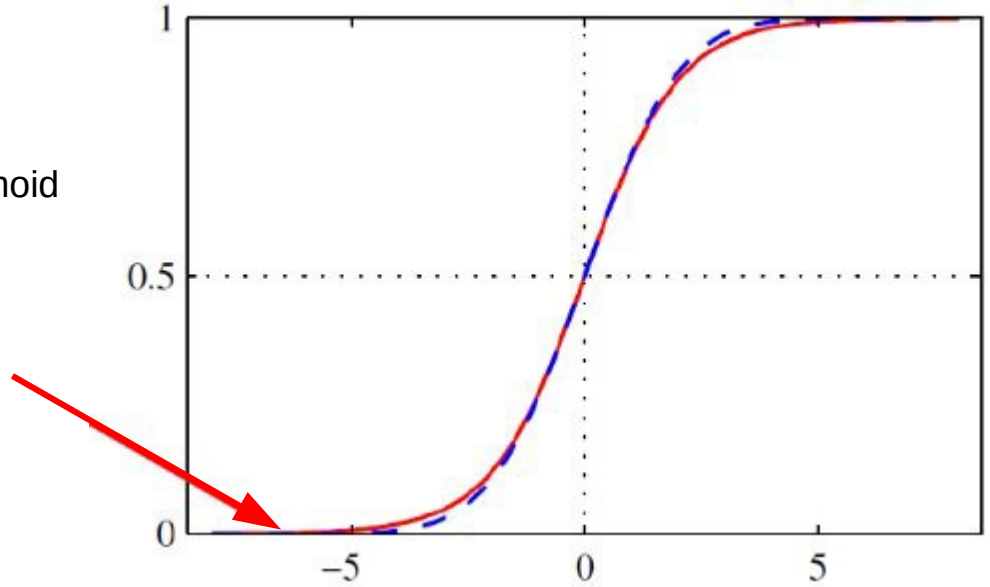
VANISHING GRADIENTS

- Deep learning community often deals with two types of problems during training: vanishing gradients (and exploding) gradients.
  - Vanishing gradients
    - the backpropagation algorithm, which chains the gradients together when computing the error backwards, will find really small gradients towards the left side of the network (i.e., farthest from where error computation started).
    - This problem primarily occurs e.g. with the Sigmoid and Tanh activation functions, whose derivatives produce outputs of $0 < x' < 1$, except for Tanh which produces $x' = 1$ at $x = 0$.
    - Consequently, when using Tanh and Sigmoid, you risk having a suboptimal model that might possibly not converge due to vanishing gradients.
    - ReLU does not have this problem – its derivative is 0 when $x < 0$ and is 1 otherwise.
    - It is computationally faster. Computing this function – often by simply maximizing between (0, x) – takes substantially fewer resources than computing e.g. the sigmoid and tanh functions. By consequence, ReLU is the de facto standard activation function in the deep learning community today.

VANISHING GRADIENTS

Problem with Sigmoid
→ Saturation

Gradient too small



VANISHING GRADIENTS

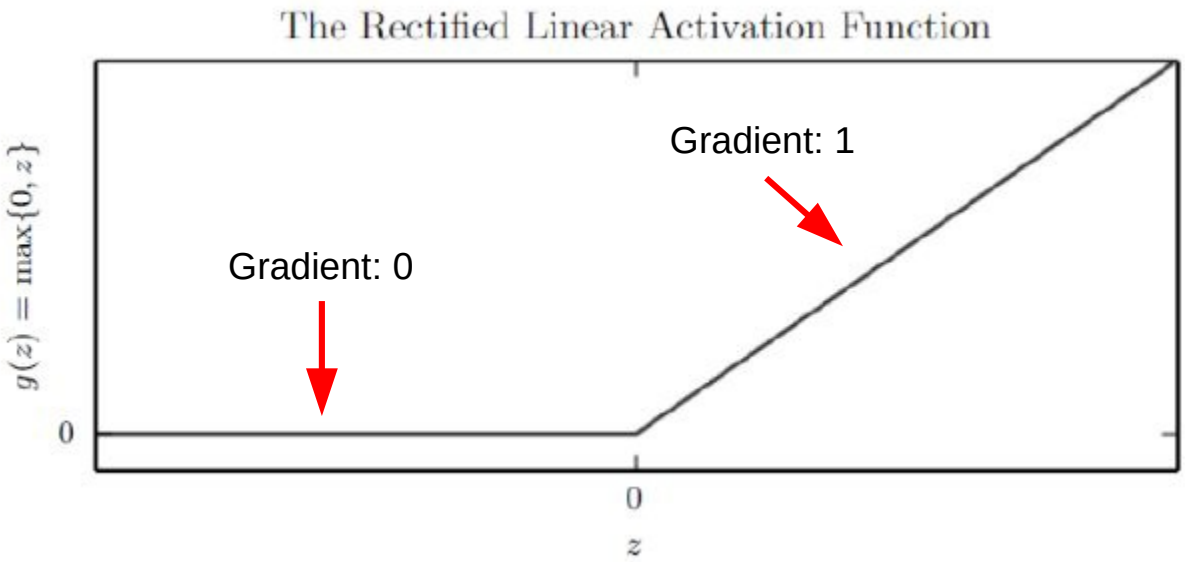
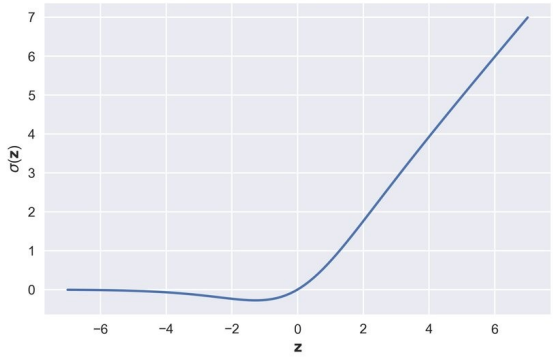


Fig. From Goodfellow et al. (2016)

SWISH ACTIVATION FUNCTION

- Nevertheless, it does not mean that it cannot be improved.
  - Swish activation function.
  - Instead, it does look like the de-facto standard activation function, with one difference: the domain around 0 differs from ReLU.
- Swish is a smooth function. That means that it does not abruptly change direction like ReLU does near x = 0.
  - Swish is non-monotonic. It thus does not remain stable or move in one direction, such as ReLU.
  - It is in fact this property which separates Swish from most other activation functions, which do share this monotonicity.
- In applications - Swish could be better than ReLU.

$$f(x) = x * \text{sigmoid}(x)$$
$$= x * \left(1 + e^{-x}\right)^{-1}$$



See, P Ramachandran, B Zoph, Q V. Le (2017) - https://arxiv.org/pdf/1710.05941v1.pdf

INTERMEZZO – ACTION REQUIRED

- Let’s look at this notebook:
- https://jupyter.cscs.ch
  → 01_GradientDescent_and_StochasticGradientDescent.ipynb

A GEOMETRIC INTERPRETATION

- In 3D, the following mental image may prove useful. Imagine two sheets of colored paper: one red and one blue.
- Put one on top of the other.
- Crumple them together into a small ball. That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem.
- What a neural network (or any other machine-learning model) is meant to do is figure out a **transformation of the paper** ball that would uncrumple it, so as to make the two classes cleanly separable again.
- With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.

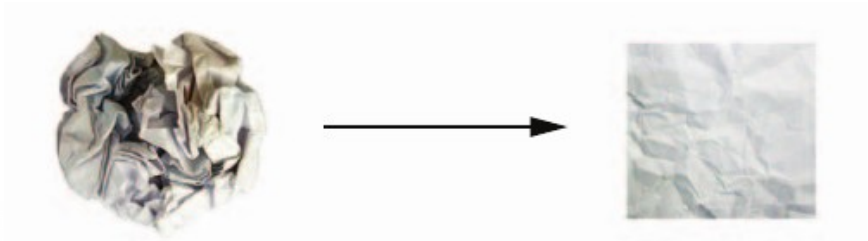
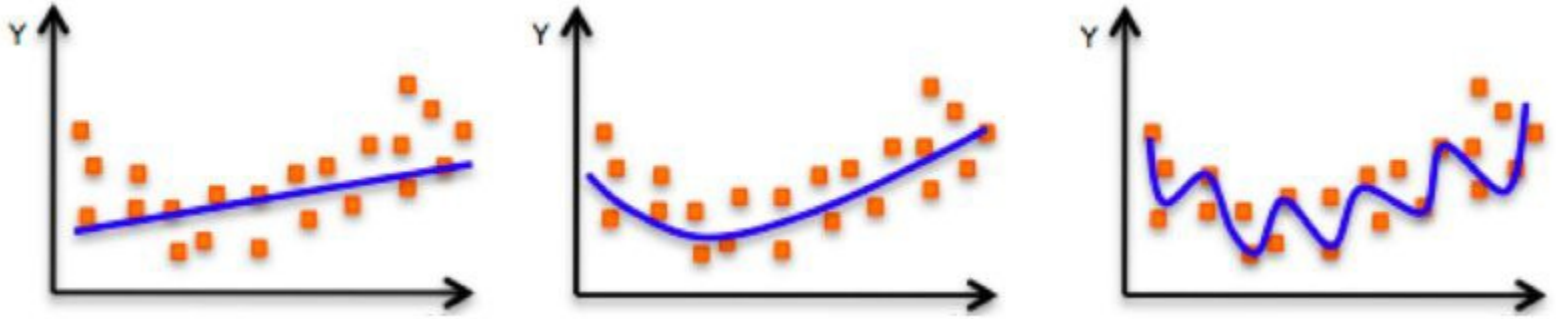


Figure 2.9 Uncrumpling a complicated manifold of data

Fig. from Chollet 2017

NOTES ON OVERFITTING



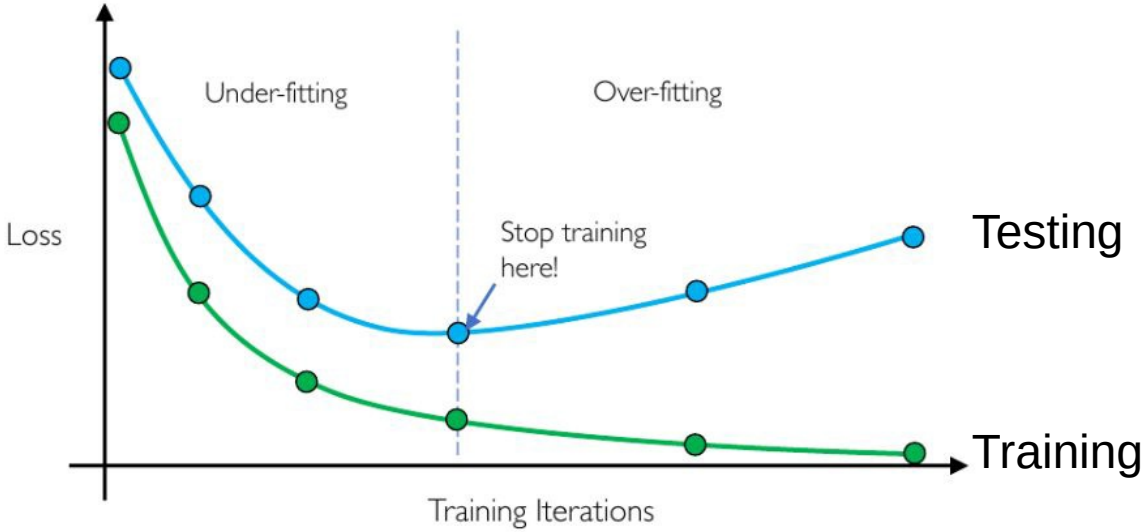
Underfitting
Model does not have capacity to fully learn the data

Ideal Fit

Overfitting
Too complex, extra parameters, does not generalize well

EARLY STOPPING

- Stop training before we have a chance to overfit

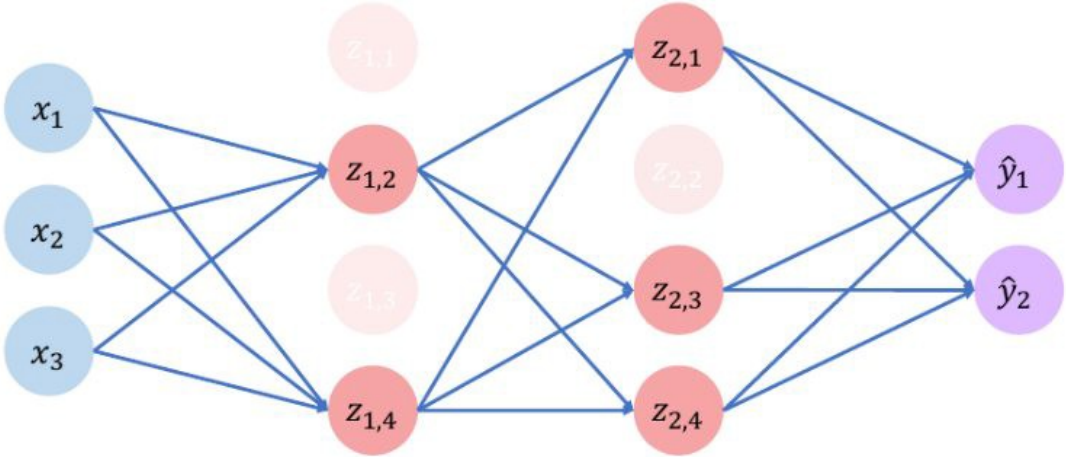


NOTES ON REGULARIZATION

- Regularization is a technique that constrains our optimization problem to discourage complex models.
- We use it to improve generalization of our model on unseen data.

REGULARIZATION IN NN: DROPOUT

- During training, randomly set some activations to 0
  - Typically 'drop' 50 % of activations in layer
  - Forces network to not rely on any node



Srivastava et al., 2014 – http://jmlr.org/papers/v15/srivastava14a.html

REGULARIZATION IN NN: DROPOUT

- It is an efficient way of performing model averaging with neural networks.
- Can be interpreted as some sort of bagging.
- Now, we assume that the model's role is to output a probability distribution. In the case of bagging, each model i produces a probability distribution $p^{(i)}(y \mid x)$.
- The prediction of the ensemble is given by the arithmetic mean of all these distributions:

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y \mid \boldsymbol{x})$$

- In the case of dropout, each sub-model defined by mask vector μ defines a probability distribution p(y | x, μ).
- The arithmetic mean over all masks is given by $\sum_{\mu} p(\boldsymbol{\mu}) p(y \mid \boldsymbol{x}, \boldsymbol{\mu})$ where p(μ) is the probability distribution that was used to sample μ at training time.

REMARK: BATCH NORMALIZATION

- https://arxiv.org/abs/1502.03167
- Batch normalization is used to stabilize and perhaps accelerate the learning process.
- It does so by applying a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.
  - Suppose we built a neural network with the goal of classifying gray-scale images. The intensity of every pixel in a gray-scale image varies from 0 to 255. Prior to entering the neural network, every image will be transformed into a 1 dimensional array. Then, every pixel enters one neuron from the input layer. If the output of each neuron is passed to a sigmoid function, then every value other than 0 (i.e. 1 to 255) will be reduced to a number close to 1. Therefore, it's common to normalize the pixel values of each image before training. Batch normalization, on the other hand, is used to apply normalization to the output of the hidden layers.

RECIPE FOR USING MLP

- Select inputs and outputs for your problem
  - Before anything else, you need to think about the problem you are trying to solve, and make sure that you have data for the problem, both input vectors and target outputs.
  - At this stage you need to choose what features are suitable for the problem and decide on the output encoding that you will use — standard neurons, or linear nodes.
  - These things are often decided for you by the input features and targets that you have available to solve the problem.
  - Later on in the learning it can also be useful to re-evaluate the choice by training networks with some input feature missing to see if it improves the results at all.

RECIPE FOR USING MLP (II)

- **Normalize inputs**
  - Re-scale the data by subtracting the mean value from each element of the input vector, and divide by the variance (or alternatively, either the maximum or minus the minimum, whichever is greater).

- **Split the data into training, testing, and validation sets**
  - You cannot test the learning ability of the network on the same data that you trained it on, since it will generally fit that data very well (often too well, over-fitting and modeling the noise in the data as well as the generating function).

  - Recall: we generally split the data into three sets, one for training, one for testing, and then a third set for validation, which is testing how well the network is learning during training.

RECIPE FOR USING MLP (III)

- **Select a network architecture**
  - You already know how many input nodes there will be, and how many output neurons.
  - You need to consider whether you will need a hidden layer at all, and if so how many neurons it should have in it.
  - You might want to consider more than one hidden layer.
  - The more complex the network, the more data it will need to be trained on, and the longer it will take.
  - It might also be more subject to over-fitting.
  - The usual method of selecting a network architecture is to try several with different numbers of hidden nodes and see which works best.

RECIPE FOR USING MLP (IV)

- **Train a network**
  - The training of the NN consists of applying the MLP algorithm to the training data.
  - This is usually run in conjunction with early stopping, where after a few iterations of the algorithm through all of the training data, the generalization ability of the network is tested by using the validation set.
  - The NN is very likely to have far too many degrees of freedom for the problem, and so after some amount of learning it will stop modeling the generating function of the data, and start to fit the noise and inaccuracies inherent in the training data. At this stage the error on the validation set will start to increase, and learning should be stopped.

- **Test the network**
  - Once you have a trained network that you are happy with, it is time to use the test data for the first (and only) time. This will enable you to see how well the network performs on some data that it has not seen before, and will tell you whether this network is likely to be usable for other data, for which you do not have targets.

KERAS & TENSORFLOW BASICS

- tensorflow.org 

- Keras API:
  https://www.tensorflow.org/guide/keras/sequential_model

- Fun data sets to play with: https://www.kaggle.com/datasets

- Some “clean” data to play with: https://archive.ics.uci.edu/ml/index.php

- Help for debugging – Tensorboard: https://www.tensorflow.org/tensorboard

A GENTLE FIRST EXAMPLE

- Lets look at the notebook: ***02_Gentle_DNN.ipynb***.
- This Notebook contains all the basic functionality from a theoretical point of view.
- 2 simple examples, one regression, and one classification.

ACTION REQUIRED

Look at the test functions below*. Pick three of those test functions (from Genz 1987).

- Approximate a 2-dimensional function stated below with Neural Nets based 10, 50, 100, 500 points randomly sampled from $[0 , 1]^2$. Compute the average and maximum error.
- The errors should be computed by generating 1,000 uniformly distributed random test points from within the computational domain.
- Plot the maximum and average error as a function of the number of sample points.
- Repeat the same for 5-dimensional and 10-dimensional functions. Is there anything particular you observe?

$$\text{oscillatory:}\quad f_1(x) = \cos\left(2\pi w_1 + \sum_{i=1}^{d} c_i x_i\right),$$

$$\text{product peak:}\quad f_2(x) = \prod_{i=1}^{d} \left(c_i^{-2} + (x_i - w_i)^2\right)^{-1},$$

$$\text{corner peak:}\quad f_3(x) = \left(1 + \sum_{i=1}^{d} c_i x_i\right)^{-(d+1)},$$

$$\text{Gaussian:}\quad f_4(x) = \exp\left(-\sum_{i=1}^{d} c_i^2 \cdot (x_i - w_i)^2\right),$$

$$\text{continuous:}\quad f_5(x) = \exp\left(-\sum_{i=1}^{d} c_i \cdot |x_i - w_i|\right),$$

$$\text{discontinuous:}\quad f_6(x) = \begin{cases} 0, & \text{if } x_1 > w_1 \text{ or } x_2 > w_2, \\ \exp\left(\sum_{i=1}^{d} c_i x_i\right), & \text{otherwise.} \end{cases}$$

Varying test functions can be obtained by altering the parameters $c = (c_1, \ldots, c_n)$ and $w = (w_1, \ldots, w_n)$. We chose these parameters randomly from $[0, 1]$. Similarly to Barthelmann et al. [2000], we normalized the c_i such that $\sum_{i=1}^{d} c_i = b_j$, with b_j depending on d, f_j according to

j	1	2	3	4	5	6
b_j	1.5	d	1.85	7.03	20.4	4.3

Furthermore, we normalized the w_i such that $\sum_{i=1}^{d} w_i = 1$.

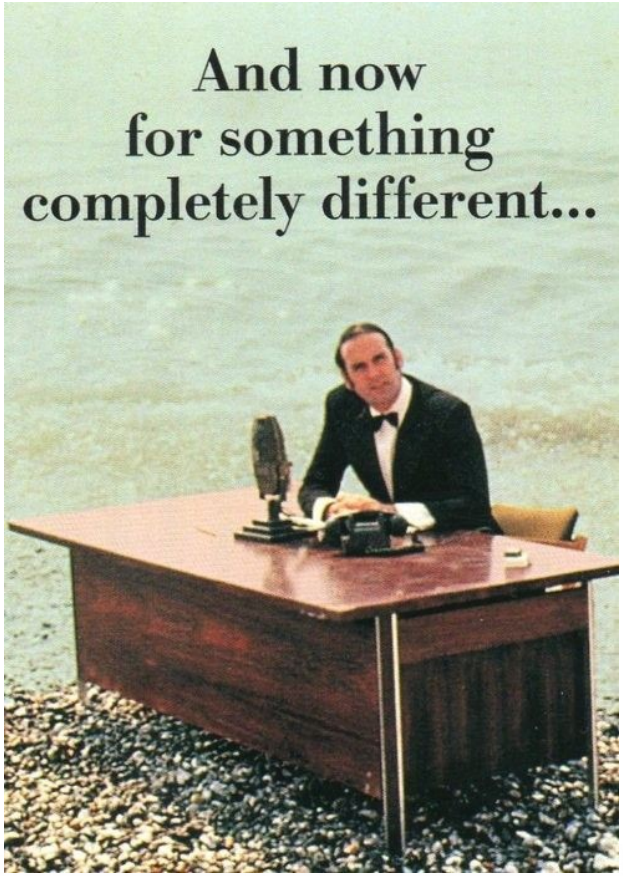
*Choose the parameters w and c in meaningful ways.

ACTION REQUIRED (II)

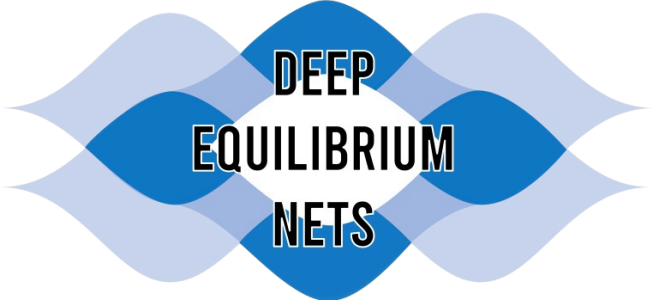
- Play with the architecture.
  - Number of hidden layers.
  - activation functions.
  - choice of the stochastic gradient descent algorithm.
  - Monitor the performance with respect to the architecture.

SOME PERSONAL TAKE-AWAY

- Swish activation is the “best” if you need smooth and deep.
- Multiple of 2 for network (training speed).
- Smaller learning rate with deeper networks.
- Batch normalization for speed.
- Glorot initialization.
- Custom layers for custom models (https://www.tensorflow.org/tutorials/customization/custom_layers).


And now
for something
completely different...

DEEP EQUILIBRIUM NETS



- Joint work with M. Azinovic, L. Gaegauf
  https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3393482

- Code examples: https://github.com/sischei/DeepEquilibriumNets

- We introduce the concept of **Deep Equilibrium Nets.**
  - **Neural networks** that directly approximate all equilibrium functions in discrete-time dynamic stochastic economic models.
  - They are trained in an **unsupervised** fashion to satisfy all equilibrium conditions along simulated paths of the economy.

- Neural network approximates the equilibrium functions directly

  → neither sets of non-linear equations nor optimization problems need to be solved in order to simulate the economy.

  → **Training data can be generated at virtually zero cost.**

  → **Generic, scalable & flexible global solution method.**

DEEP EQUILIBRIUM NETS

- Solving e.g., rich OLG models globally numerically is a formidable task.
  - Models are often formulated in a **stylized** fashion to remain computationally **tractable**.
  - We develop a **generic** solution framework to solve models of unprecedented complexity.

- **Key ideas:**
  - Use the definition of the equilibrium functions, i.e., the **implied error in the optimality conditions**, as **loss function**.
  - Learn the equilibrium functions with stochastic gradient descent.
  - Take the (training) data points from a simulated path (can be generated at virtual zero cost).

HOW TO FIND GOOD PARAMETERS OF THE NETWORK?

- The deeper and larger the neural net becomes, the more flexible it is as a function approximator, …

- ... but the more data it will need
  - rule of thumb: **#observations/parameters** ~ 10x (Marsland (2014).

HOW TO FIND GOOD PARAMETERS OF THE NETWORK?

- In the standard way of solving for equilibria via policy iteration, we usually have **"small" data** (e.g., Sparse Grids (Brumm & Scheidegger (2017), Krüger & Kübler (2004), Judd et al. (2014))

- **Time Iteration – Collocation** (see, e.g., Judd (1998), and references therein)

1. Select a grid G, and a policy function f^{start}. Set $f^{\text{next}} \equiv f^{\text{start}}$.
2. Make one time iteration step:
    1. For all $g \in G$, find $f(g)$ that solves the Period-to-Period Equilibrium Problem given f^{next}.
    2. Use solutions at all grid points $G = G$ to interpolate f(how?).
3. Check error criterion: If $\|f - f^{\text{next}}\|_\infty < \epsilon$, report solution: $\tilde{f} = f$ Else set $f^{\text{next}} \equiv f$ and go to step 2.

DEEP EQUILIBRIUM NETS

- Define an economic loss-function $$l_\rho := \frac{1}{N_{\text{path length}}} \sum_{\text{x}_\text{i} \text{ on sim. path}} \left(\mathbf{G}\left(\mathbf{x}_i, \mathcal{N}_\rho\left(\mathbf{x}_i\right)\right)\right)^2$$

  where we use $\mathcal{N}_\rho$ to simulate a path.

- G is chosen such that the true equilibrium policy f(x) is defined by
  - G(x, f(x)) = 0 ∀x.
- G(.,.): implied error in the optimality conditions (unit-free Euler errors)
- Therefore, there is no need for labels to evaluate our loss function.
  - Unsupervised Machine Learning.

DEEP EQUILIBRIUM NETS

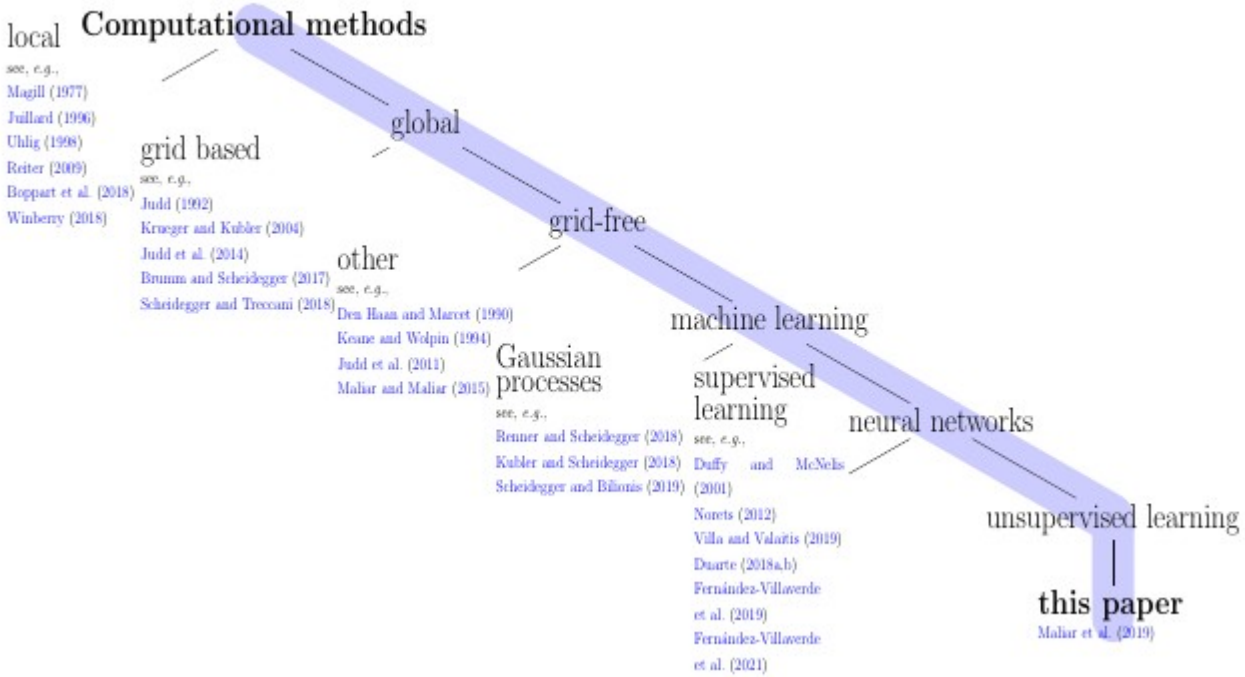
Algorithm 1: Algorithm for training deep equilibrium nets.

Data:
T (length of an episode),
N^{epochs} (number of epochs on each episode),
$\tau^{\max}$ (desired threshold for max error),
τ^{mean} (desired threshold for mean error),
$\epsilon^{\text{mean}} = \infty$ (starting value for current mean error),
$\epsilon^{\max} = \infty$ (starting value for current max error),
N^{iter} (maximum number of iterations),
$\boldsymbol{\rho}^0$ (initial parameters of the neural network),
$\mathbf{x}_1^0$ (initial state to start simulations from),
$i = 0$ (set iteration counter),
α^{learn} (learning rate)
Result:
success (boolean if thresholds were reached)
$\boldsymbol{\rho}^{\text{final}}$ (final neural network parameters)
while $((i < N^{iter}) \wedge ((\epsilon^{mean} \geq \tau^{mean}) \vee (\epsilon^{max} \geq \tau^{max})))$ **do**
  $\mathcal{D}^i_{\text{train}} \leftarrow \{\mathbf{x}_1^i, \mathbf{x}_2^i, \ldots, \mathbf{x}_T^i\}$ (generate new training data by simulating an episode of T periods as implied by the parameters $\boldsymbol{\rho}^i$)
  $\mathbf{x}_0^{i+1} \leftarrow \mathbf{x}_T^i$ (set new starting point)
  $\epsilon_{\max} \leftarrow \max\left\{\max_{\mathbf{x}\in\mathcal{D}^i_{\text{train}}} |e_{\mathbf{x}}(\boldsymbol{\rho})|\right\}$ (calculate max error on new data)
  $\epsilon_{\text{mean}} \leftarrow \max\left\{\frac{1}{T}\sum_{\mathbf{x}\in\mathcal{D}^i_{\text{train}}} |e_{\mathbf{x}}(\boldsymbol{\rho})|\right\}$ (calculate mean error on new data)
  for $j \in [1, ..., N^{epochs}]$ **do**
    (learn N^{epochs} on data)
    for $k \in [1, ..., length(\boldsymbol{\rho})]$ **do**

$$\rho_k^{i+1} = \rho_k^i - \alpha^{\text{learn}} \frac{\partial \ell_{\mathcal{D}^i_{\text{train}}}(\boldsymbol{\rho}^i)}{\partial \rho_k^i} \tag{54}$$

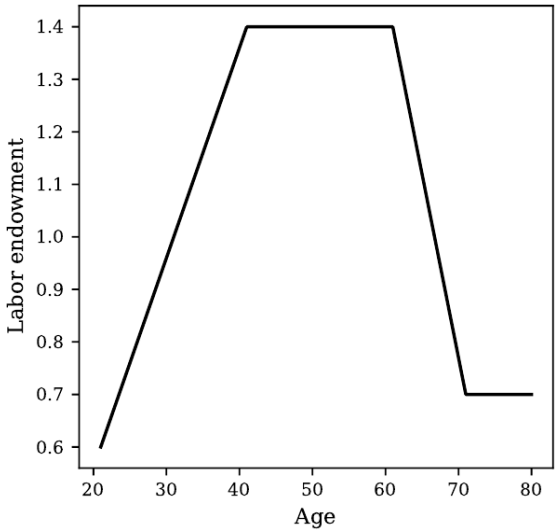
      (do a gradient descent step to update the network parameters)
    end
  end
  $i \leftarrow i + 1$ (update episode counter)
end
if $i = N^{iter}$ **then return** (success $\leftarrow False, \boldsymbol{\rho}^{\text{final}} \leftarrow \boldsymbol{\rho}^i$) ;
else return (success $\leftarrow True, \boldsymbol{\rho}^{\text{final}} \leftarrow \boldsymbol{\rho}^i$) ;

AN INCOMPLETE LIST OF RELATED LITERATURE



BENCHMARK OLG MODEL AS EXAMPLE

- Time is discrete: $t = 0, \ldots, \infty$
- Agents live for *N* periods (N=60 years).
- One representative household per cohort.
- Every *t*, a representative household in born.
- No uncertainty about lifetime.
- There are exogenous aggregate shocks *z* that follow a Markov chain.
- Each period, the agents alive receive a strictly positive labor endowment which depends on the age of the agent alone.



HOUSEHOLDS

- Household supplies its labour endowment inelastically for a market wage w_t.

- Agents alive maximize their remaining time-separable discounted expected lifetime utility ($\beta < 1$):

$$\sum_{i=0}^{N-s} \mathrm{E}_t \left[\beta^i u \left(c_{t+i}^{s+i} \right) \right]$$

- Households can save a unit of consumption good to obtain a unit of capital good next period (denoted as a_t^s).

- The savings will become capital in the next period:

$$a_t^s = k_{t+1}^{s+1}, \forall t, \forall s \in \{1, \ldots, N-1\}$$

HOUSEHOLDS (II)

- Households cannot die with debt.
- Borrowing is allowed up to an exogenously given level: $a_t^s \geq \underline{a}$
- At time *t*, the households sell their capital to the firm at market price $\mathbf{r}_t > 0$.
- The budget constraint of the household **s** in period *t* is

$$c_t^s + a_t^s = r_t k_t^s + l_t^s w_t$$

- The agents are born, and die without any assets $k_t^1 = 0 \text{ and } a_t^N = 0$

FIRMS AND MARKETS

- There is a single representative firm with Cobb-Douglas production.

- The total factor productivity η (TFP) and the depreciation δ depend on the exogenous shock *z* alone (η(z) ∈ {0.85, 1.15}, δ(z) ∈ {0.5, 0.9})

$$\pi^{\delta} = \begin{bmatrix} 0.98 & 0.02 \\ 0.25 & 0.75 \end{bmatrix}, \qquad \pi^{\eta} = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} \qquad z^{\delta} \otimes z^{\eta} = z \in \{0, 1, 2, 3\}$$

- Each period, after the shock has realized, the firm buys capital and hires labor to maximize its profits, taking prices as given.

- The stochastic production function is given by $f(K, L, z) = \eta(z) K^{\alpha} L^{1-\alpha} + K(1 - \delta(z))$

- There are competitive spot markets for consumption, capital, labor.

EQULIBRIUM

Definition 1 (competitive equilibrium) *A competitive equilibrium, given initial conditions* $z_0, \{k_0^s\}_{s=1}^{N-1}$, *is a collection of choices for households* $\{(c_t^s, a_t^s)_{s=1}^N\}_{t=0}^\infty$ *and for the representative firm* $(K_t, L_t)_{t=0}^\infty$ *as well as prices* $(r_t, w_t)_{t=0}^\infty$, *such that*

1. *Given* $(r_t, w_t)_{t=0}^\infty$, *the choices* $\{(c_t^s, a_t^s)_{s=1}^N\}_{t=0}^\infty$ *maximize* (1), *subject to* (2), (3), *and* (4).
2. *Given* r_t, w_t, *the firm maximizes profits,* i.e.,

$$(K_t, L_t) \in \arg\max_{K_t, L_t \geq 0} f(K_t, L_t, z_t) - r_t K_t - w_t L_t.$$

3. *All markets clear: For all* t

$$L_t = \sum_{s=1}^{N} l_t^s,$$

$$K_t = \sum_{s=1}^{N} k_t^s,$$

(1): max. remaining lifetime utility
(2): savings → capital in next period
(3): borrowing constraint.
(4): budget constraint.

FUNCTIONAL RATIONAL EXPECTATIONS EQUILBRIUM

- FREE: A function mapping states to policies that are consistent with the equilibrium conditions.

$$\mathbf{f} : \{0,1,2,3\} \times \mathbb{R}^{60} \to \mathbb{R}^{59\cdot 2} : \quad \mathbf{f}\left(\begin{bmatrix} z_t \\ \mathbf{k}_t \end{bmatrix}\right) = \mathbf{f}\left(\begin{bmatrix} z_t \\ 0_t^1 \\ k_t^2 \\ \dots \\ k_t^{59} \\ k_t^{60} \end{bmatrix}\right) = \begin{bmatrix} a_t^1 \\ \dots \\ a_t^{59} \\ \lambda_t^1 \\ \dots \\ \lambda_t^{59} \end{bmatrix}$$

capital investment funct.

kkt-multiplier borrowing contr.

such that: $\forall h = 1, \dots, 59$:

$$\left.\begin{aligned} 0 &= \beta \mathrm{E}_t\left[\frac{R_{t+1}u'(c_{t+1}^{h+1}) + \lambda_t^h}{u'(c_t^h)}\right] - 1 \\ 0 &= \lambda_t^h a_t^h \\ 0 &\le \lambda_t^h \\ 0 &\le a_t^h \end{aligned}\right\} =: \mathbf{G}\left(\begin{bmatrix} z_t \\ \mathbf{k}_t \end{bmatrix}, \mathbf{f}\left(\begin{bmatrix} z_t \\ \mathbf{k}_t \end{bmatrix}\right)\right)_h$$

$$\begin{aligned} c_t^h &= k_t^h R_t + l_t^h w_t - a_t^h \\ R_t &= \xi_t \alpha K_t^{\alpha-1} L_t^{1-\alpha} + (1-\delta_t) \\ w_t &= \xi_t (1-\alpha) K_t^{\alpha} L_t^{-\alpha} \\ K_t &= \sum_{h=1}^{60} k_t^h \\ L_t &= \sum_{h=1}^{60} l_t^h \end{aligned}$$

DEEP EQUILIBRIUM NETS

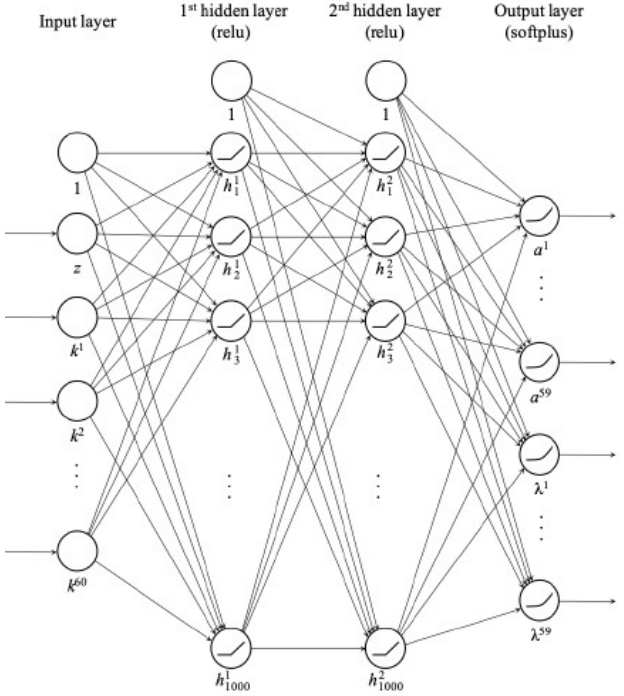
$$\mathcal{N}_{\rho} : \{0,1,2,3\} \times \mathbb{R}^{60} \to \mathbb{R}^{59 \cdot 2} :$$

$$\mathcal{N}_{\rho}\left(\begin{bmatrix} z_t \\ \mathbf{k}_t \end{bmatrix}\right) = \begin{bmatrix} a_t^1 \\ \dots \\ a_t^{59} \\ \lambda_t^1 \\ \dots \\ \lambda_t^{59} \end{bmatrix}$$

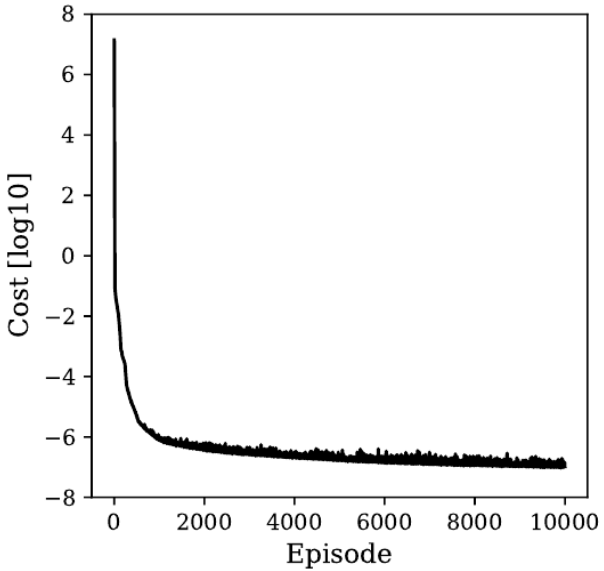
such that

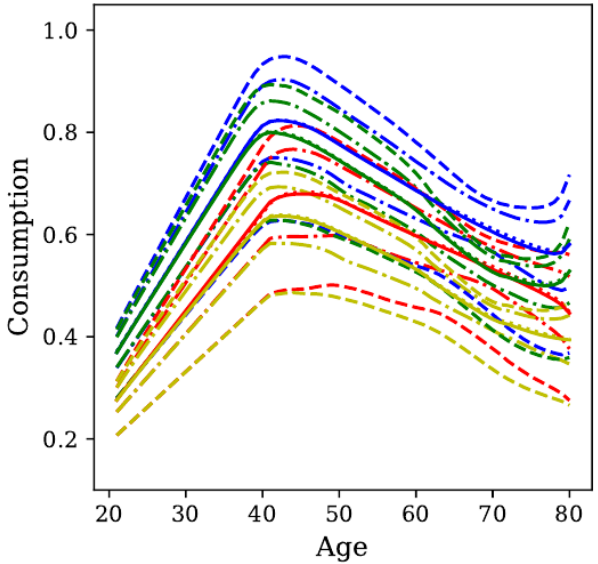
$$\mathbf{G}\left(\begin{bmatrix} z_t \\ \mathbf{k}_t \end{bmatrix}, \mathcal{N}_{\rho}\left(\begin{bmatrix} z_t \\ \mathbf{k}_t \end{bmatrix}\right)\right) \approx \mathbf{0}$$

$$\hat{\mathbf{x}}_+ = \begin{bmatrix} z_+ \\ 0 \\ \hat{a}^{[1:N-1]}(\mathbf{x}) \end{bmatrix}$$



LEARNING THE EQUILIBRIUM



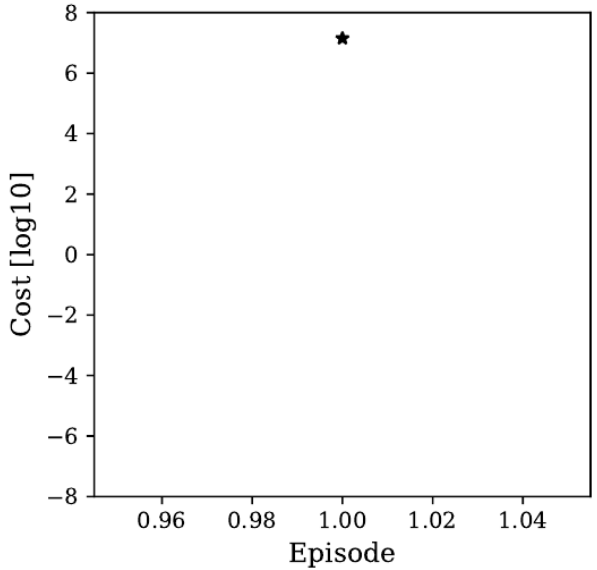


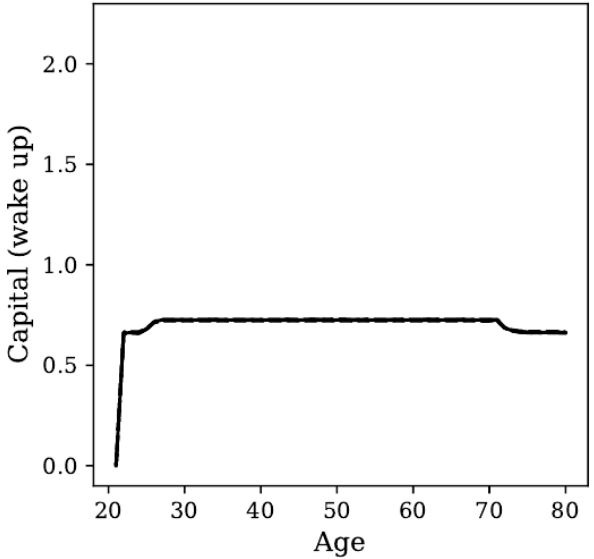
shock 1 shock 2 shock 3 shock 4

shock 1: $\delta = 0.5,\ \xi = 0.85$, shock 2: $\delta = 0.5,\ \xi = 1.15$, shock 3: $\delta = 0.9,\ \xi = 0.85$, shock 4: $\delta = 0.9,\ \xi = 1.15$

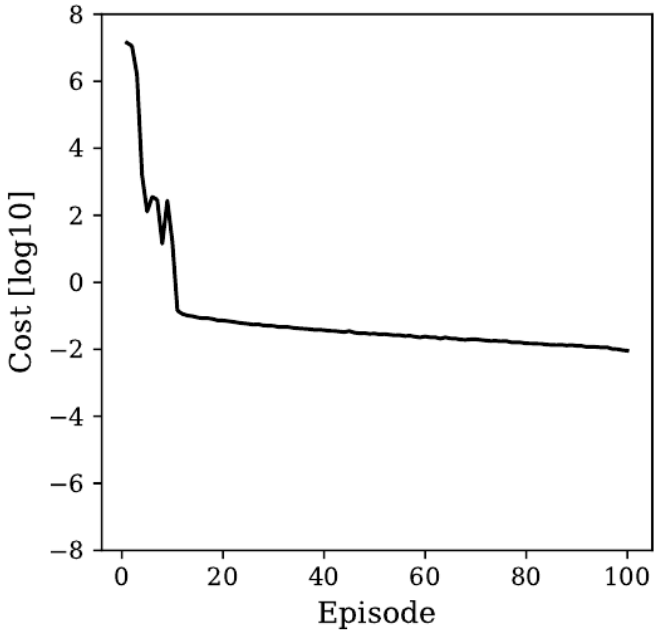
LEARNING THE EQUILIBRIUM

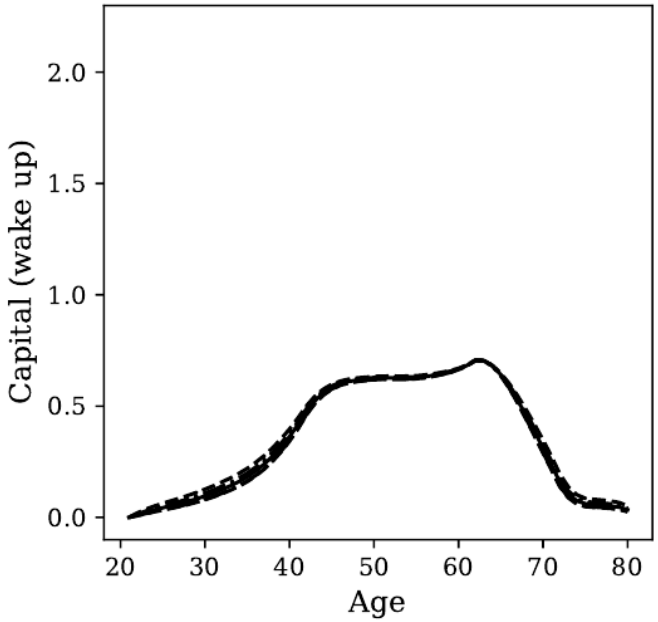
Def.: **Episode**: the set of T simulated periods.
Epoch: when the whole dataset is passed through the algorithm.
Per epoch, the neural network parameters are updated T /m times.





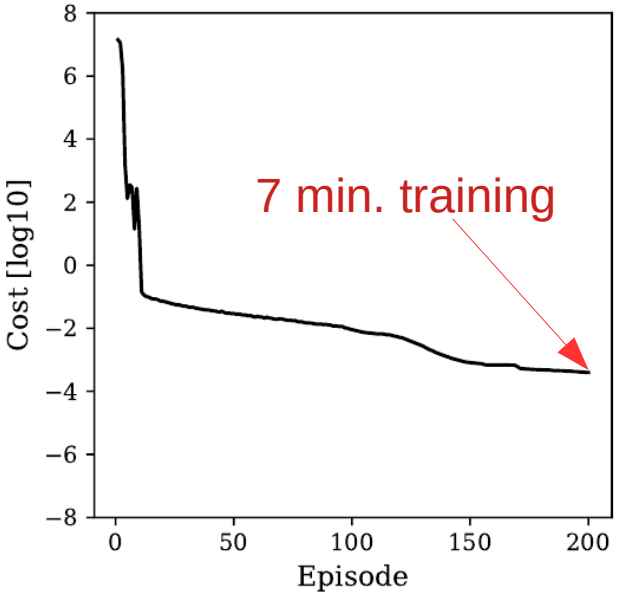
LEARNING THE EQUILIBRIUM

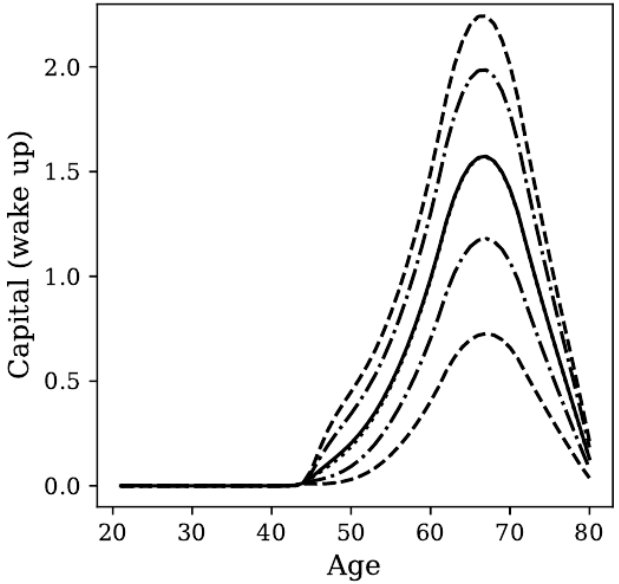




LEARNING THE EQUILIBRIUM

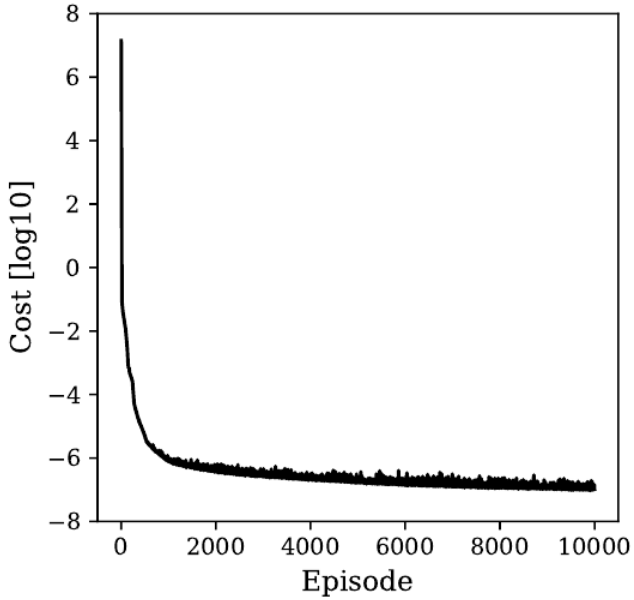
Solid line: **mean** over 10,000 simulated periods
Dash-dotted lines show the **10th and 90th percentile**
Dashed lines show the **0.1th and 99.9th percentile**.

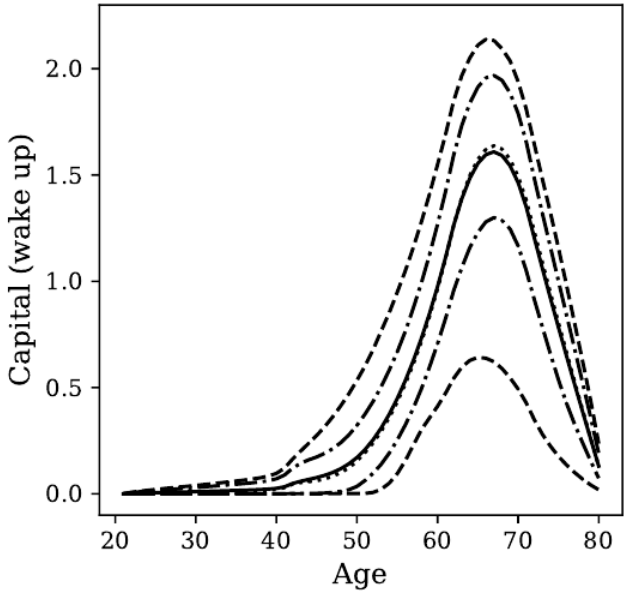




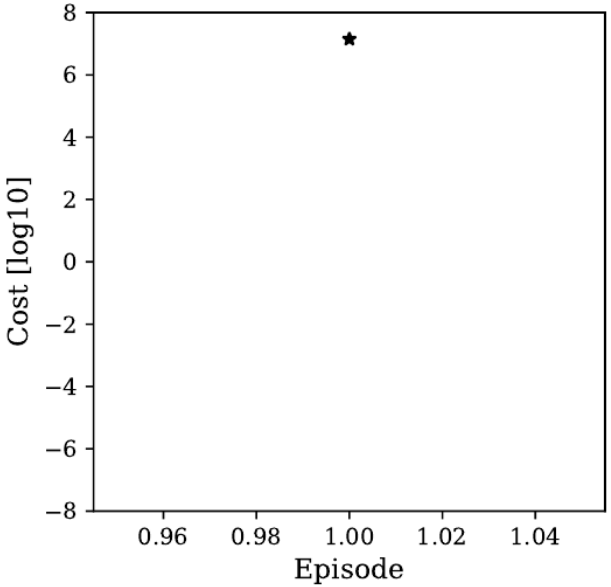
LEARNING THE EQUILIBRIUM

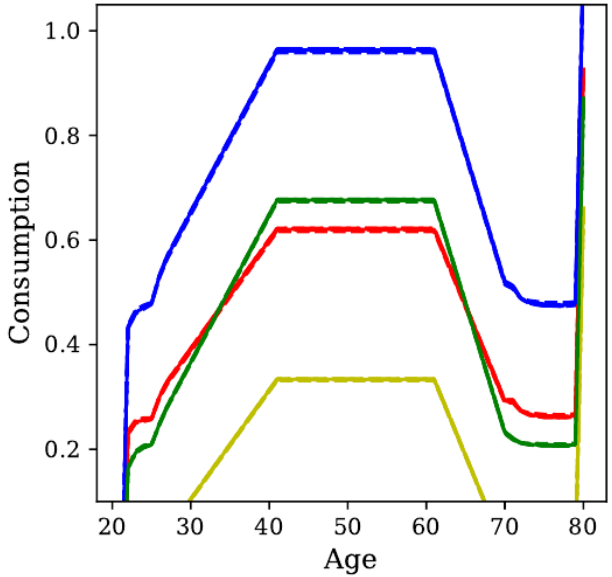
Solid line: **mean** over 10,000 simulated periods
Dash-dotted lines show the **10th and 90th percentile**
Dashed lines show the **0.1th and 99.9th percentile**.





LEARNING THE EQUILIBRIUM

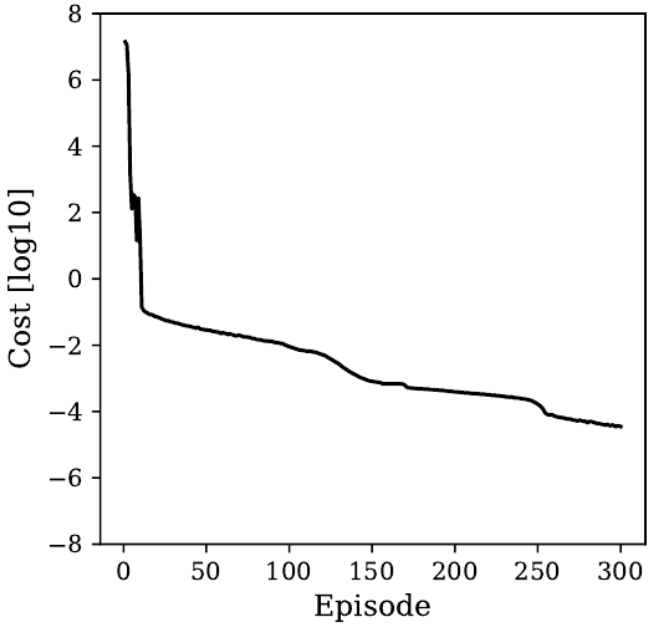


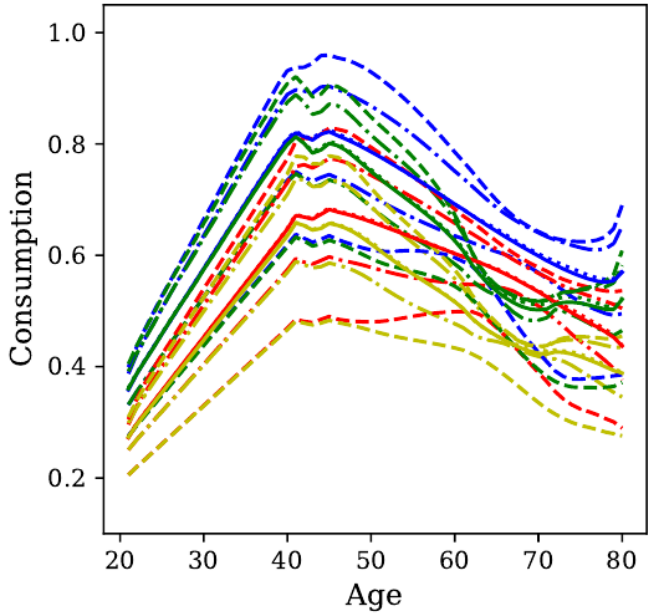


shock 1 shock 2 shock 3 shock 4

shock 1: $\delta = 0.5,\ \xi = 0.85$, shock 2: $\delta = 0.5,\ \xi = 1.15$, shock 3: $\delta = 0.9,\ \xi = 0.85$, shock 4: $\delta = 0.9,\ \xi = 1.15$

LEARNING THE EQUILIBRIUM

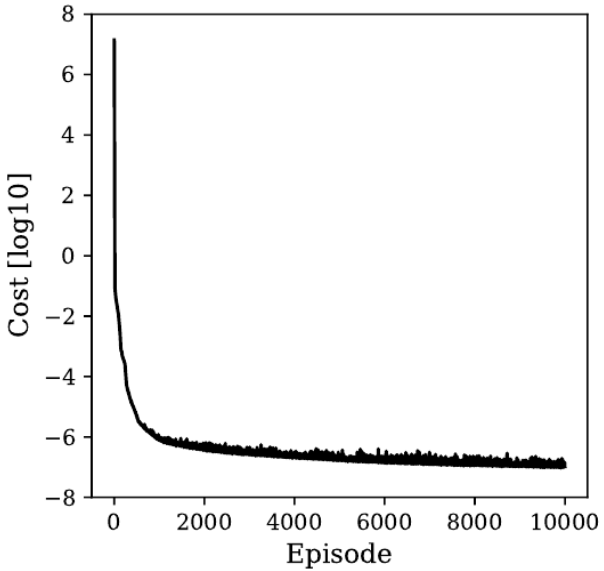


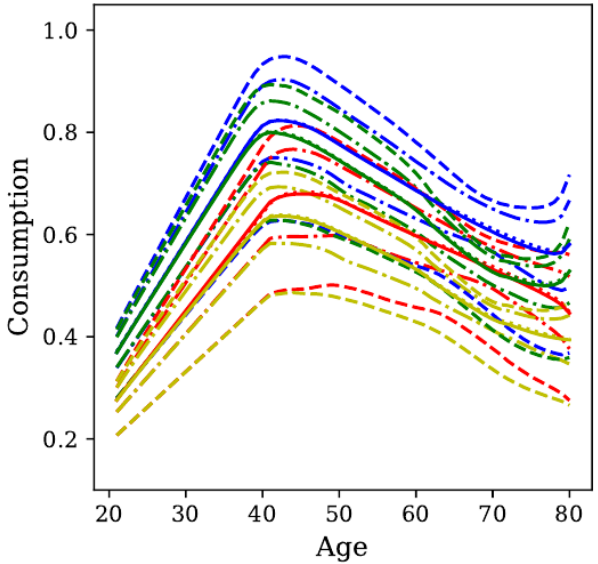


shock 1 shock 2 shock 3 shock 4

shock 1: $\delta = 0.5,\ \xi = 0.85$, shock 2: $\delta = 0.5,\ \xi = 1.15$, shock 3: $\delta = 0.9,\ \xi = 0.85$, shock 4: $\delta = 0.9,\ \xi = 1.15$

LEARNING THE EQUILIBRIUM

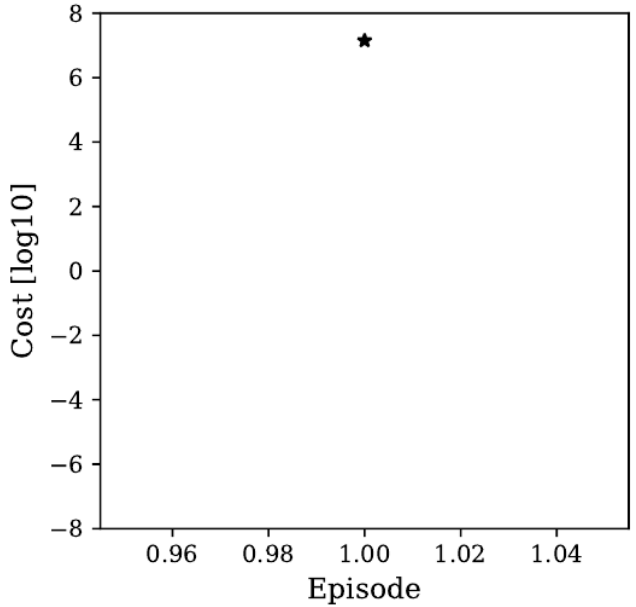


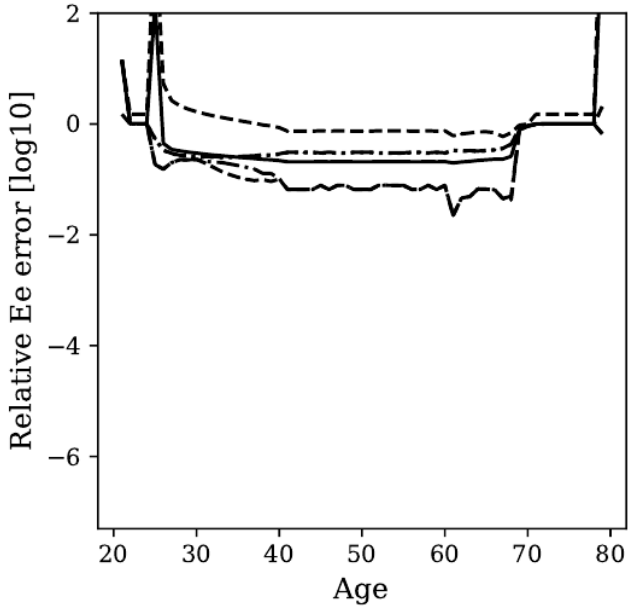


shock 1 shock 2 shock 3 shock 4

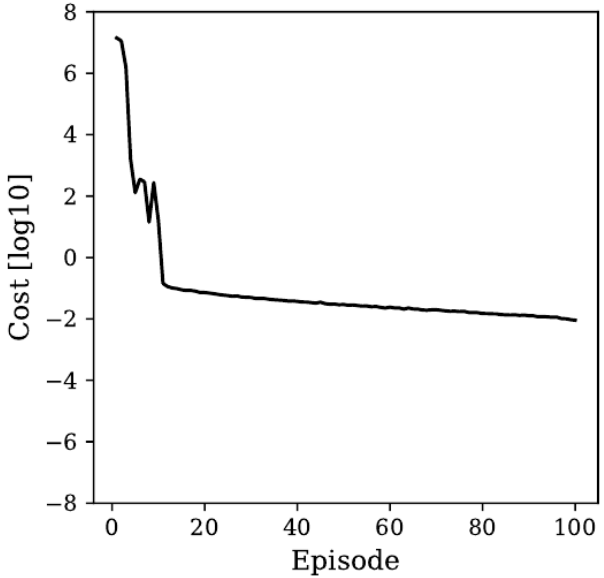
shock 1: $\delta = 0.5,\ \xi = 0.85$, shock 2: $\delta = 0.5,\ \xi = 1.15$, shock 3: $\delta = 0.9,\ \xi = 0.85$, shock 4: $\delta = 0.9,\ \xi = 1.15$

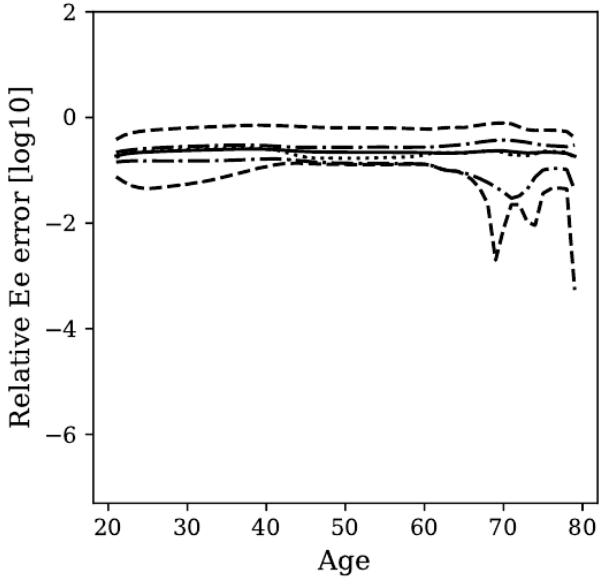
LEARNING THE EQUILIBRIUM



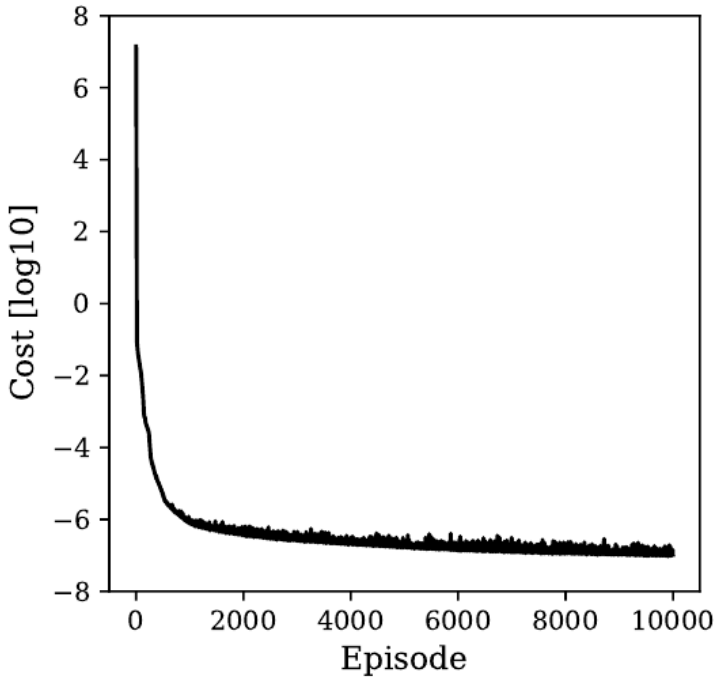


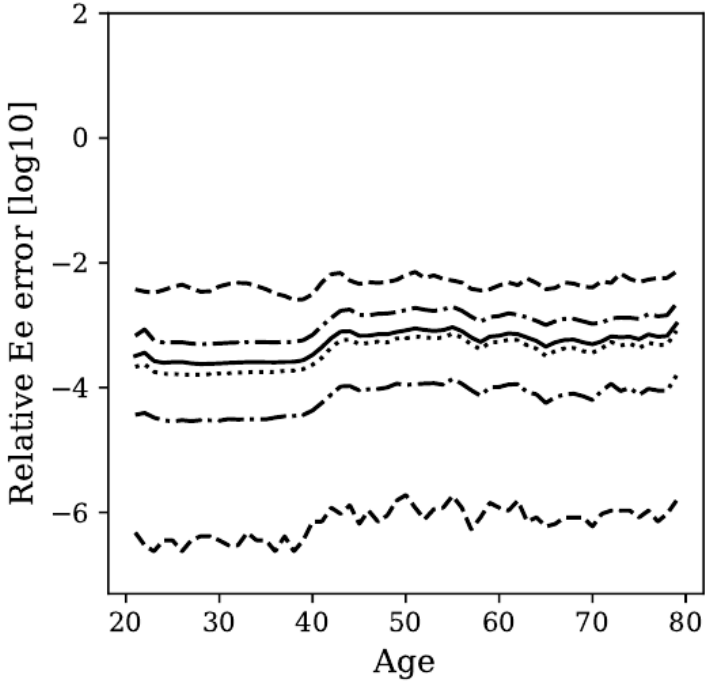
LEARNING THE EQUILIBRIUM





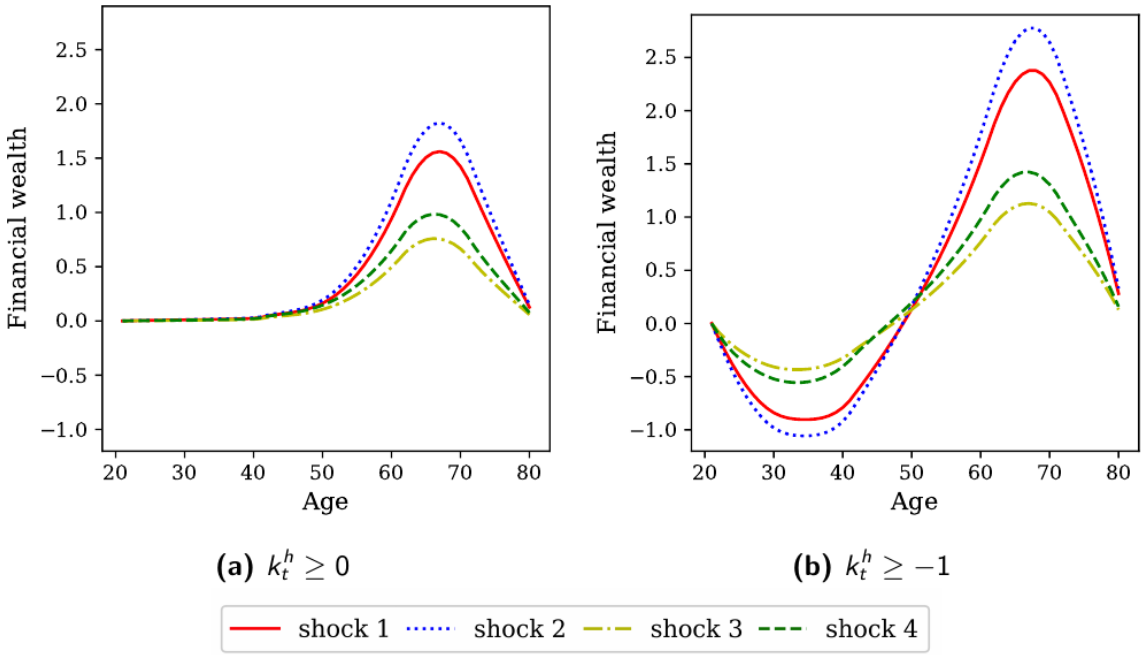
LEARNING THE EQUILIBRIUM





LOOSENING BORROWING CONSTRAINTS

Financial wealth across age-groups



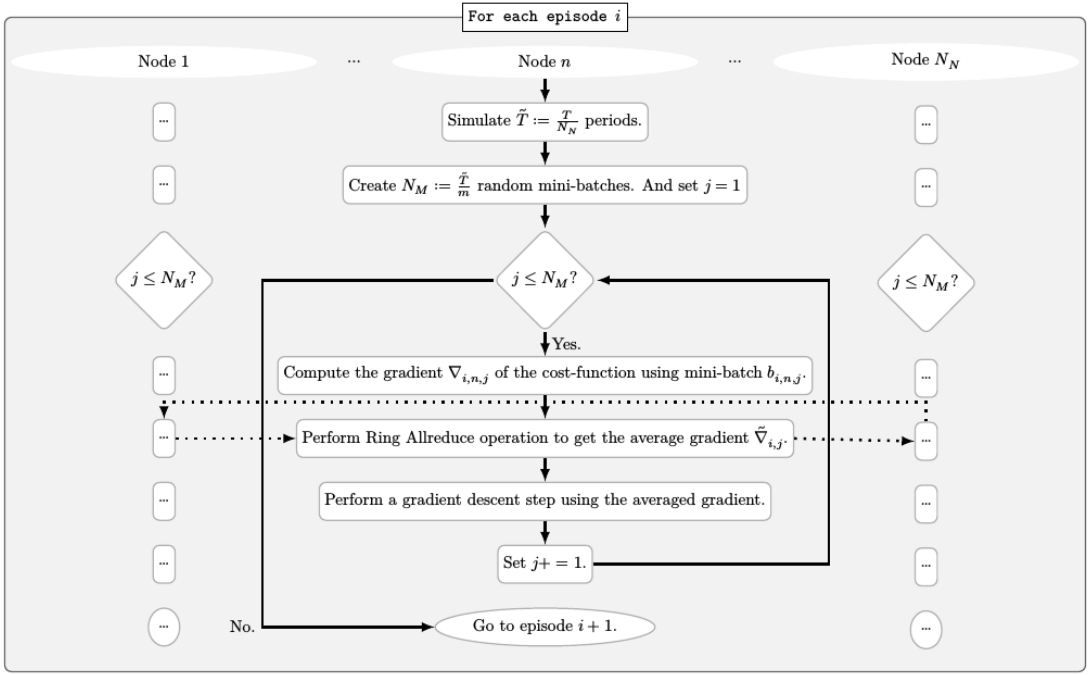
(a) $k_t^h \geq 0$ **(b)** $k_t^h \geq -1$

shock 1 : $\delta = 0.5, \xi = 0.85,$ shock 2 : $\delta = 0.5, \xi = 1.15,$ shock 3 : $\delta = 0.9, \xi = 0.85$ shock 4 : $\delta = 0.9, \xi = 1.15$

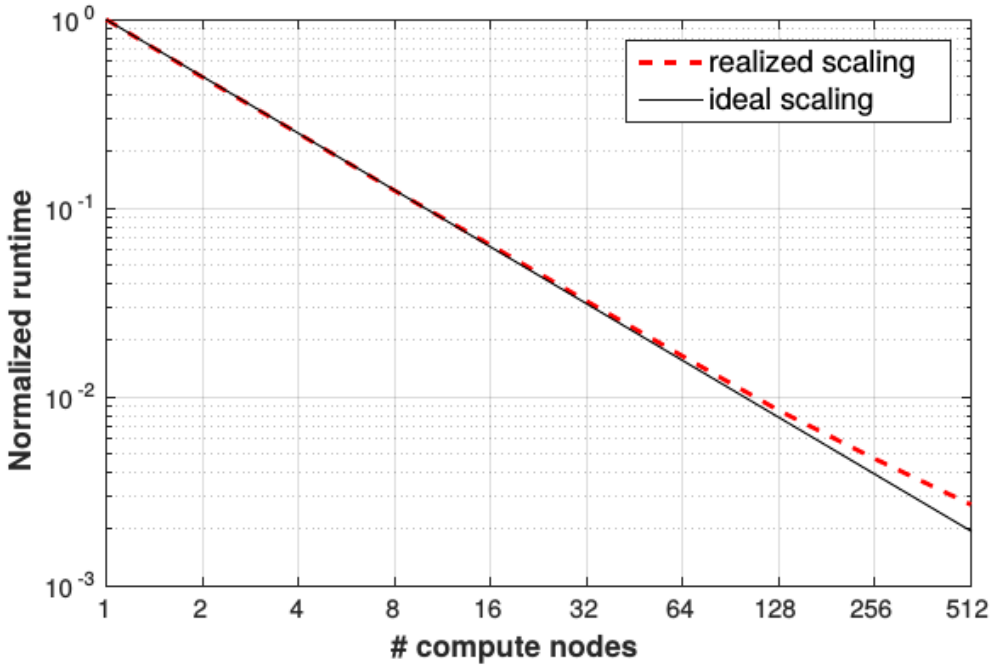
HOROVOD

- https://eng.uber.com/horovod/
- We use Horovod to parallelize DEQ.
- Based on MPI.
- Idea: **use data-parallelism**.
- The **training data is divided across nodes**.
- Each node computes the gradient of the cost-function concerning the parameters of the neural network on its given batch of data.
- Then, all nodes are synchronized and the gradient descent step is taken using the **average gradient**.
- Horovod implements the synchronization and calculation of the average gradient using a Ring Allreduce operation.

PARALLELIZATION SCHEME

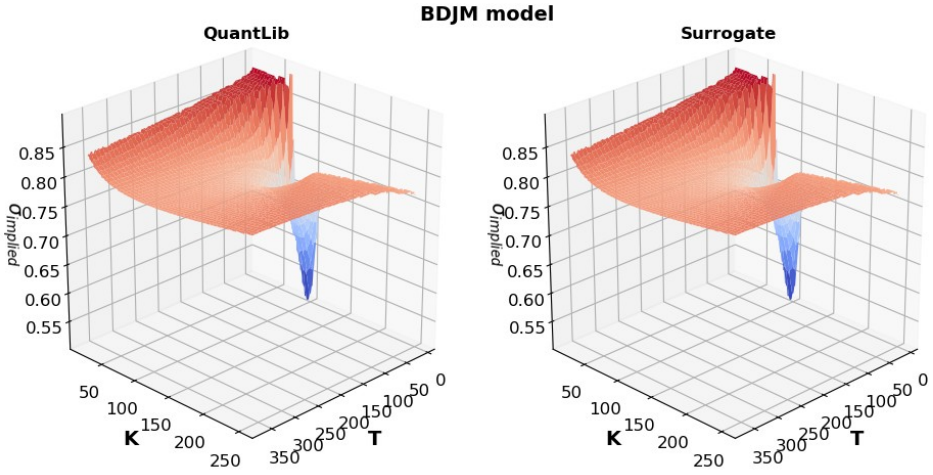


SCALABILITY



DEEP STRUCTURAL ESTIMATION

With A. Didisheim (UNIL), H. Chen (MIT)
https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3885021



MOTIVATION

- Contemporary models very rich (many endogenous states, exogenous states, strong non-linearities, lots of parameters,...).
- Expensive to compute.
- Consequently, economists are often forced to sacrifice certain features of the model in order to reduce model dimensionality
  - estimate only a partial set of parameters while prefixing the others
  - estimate the model only once using the full sample.
- The high computational costs limit a researcher's ability to carry out a variety of important model analyses.
- Model **estimation**, **calibration**, and **uncertainty quantification** can be daunting numerical tasks

  → because of the need to perform sometimes hundreds of thousands of model evaluations to obtain converging estimates of the relevant parameters and converging statistics

(see, e.g., Fernández-Villaverde, Rubio-Ramrez, and Schorfheide, 2016; Fernndez-Villaverde and Guerrn-Quintana, 2020; Iskhakov, Rust, and Schjerning, 2020; Igami, 2020, among others).

THE BASIC IDEA

- Replace the economic model with a surrogate!
- Consider a model

$$f : \mathbb{R}^m \rightarrow \mathbb{R}^k = f(\Omega_t, H_t|\Theta) = y_t$$

  - where Ω_t is a vector of dimension ω containing the observable states
  - H_t is a vector of dimension h comprising the hidden states
  - Θ is a vector of dimension θ containing model parameters
  - y_t is a vector of dimension k comprising the predicted quantities of interest

THE BASIC IDEA (II)

- The problem is that f (·) can be computationally costly, so we wish to construct a cheap to evaluate surrogate, i.e., a Neural Network that replaces the “true” function f (·):

$$\hat{f}(\Omega_t, H_t, \Theta) = \hat{f}(X_t) = y_t$$

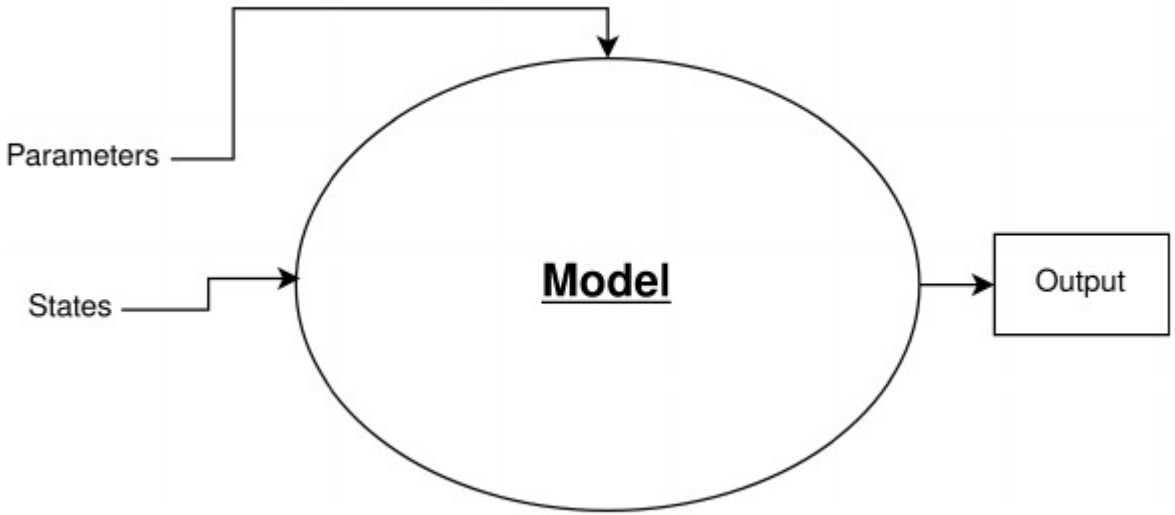
- We introduce **parameters as pseudo-state variables** (cf. Norets (2012), Scheidegger & Bilionis (2019))

  $X_t = [\Omega_t , H_t, \Theta]^T$.

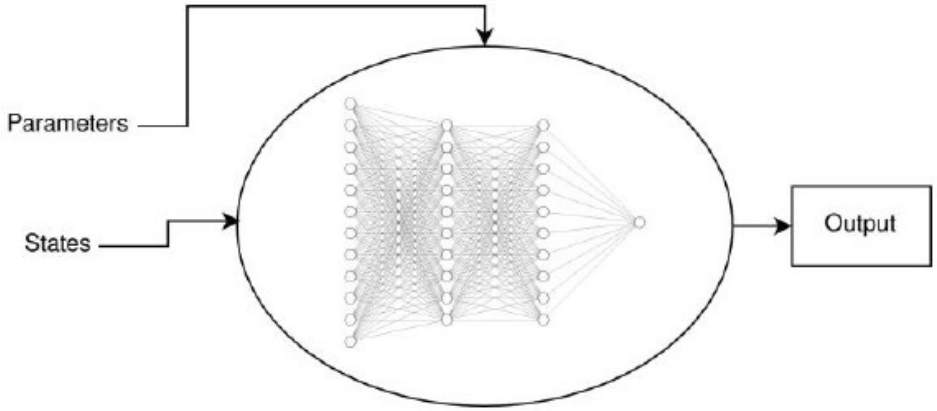
- **Solve model only once**, as a function of X_t (global solution) e.g. by using Deep Learning, e.g., by DEQN.

- For reasonable parameter ranges, you may have to use “expert knowledge”.

WHY DEEP SURROGATE



WHY DEEP SURROGATE



$$f(\text{states}, \underbrace{\text{parameters}}_{\text{Pseudo-states}}) = \text{output}$$

WHY DEEP SURROGATE?

- DNNs as universal approximators (Hornik, Stinchcombe, and White 1989)
  - Every bounded continuous function can be approximated with arbitrarily small error by network with one hidden layer.
- Sparse grids can alleviate the curse of dimensionality (Bungartz and Griebel 2004).
- Under certain conditions, the errors of approximating multivariate functions by deep ReLU networks can be bounded by sparse grids (Montanelli and Du 2019).

Cartesian Grid vs. Sparse Grid

d	$\lvert V_4 \rvert$	$\lvert V_4^S \rvert$
1	15	15
2	225	49
3	3,375	111
4	50,625	209
5	759,375	351
10	$5.77 \cdot 10^{11}$	2,001
20	$3.33 \cdot 10^{23}$	13,201

USING THE SURROGATE FOR STRUCTURAL ESTIMATION

- From time t = 1, .., T we observe N_t target states observable states,

$$\begin{aligned} \hat{Y}_t &= [\hat{y}_t^1, \hat{y}_t^2, ..., \hat{y}_t^{N_t}], \\ \hat{\Omega}_t &= [\hat{\Omega}_t^1, \hat{\Omega}_t^2, ..., \hat{\Omega}_t^{N_t}], \end{aligned}$$

- We wish to estimate Θ (parameters), and the hidden state:

$$\hat{H}_t = [\hat{H}_t^1, \hat{H}_t^2, ..., \hat{H}_t^{N_t}],$$

- With our surrogate, we can directly solve with BFGS optimization:

$$\hat{H}_1^*, \hat{H}_2^*, ...\hat{H}_T^*, \Theta^* = \underset{\hat{H}_1, \hat{H}_2, ...\hat{H}_T, \Theta}{\arg\min} \frac{1}{T} \frac{1}{N_t} \sum_{\tau=1}^{T} \sum_{i=1}^{N_t} \left(\phi(\left[\hat{\Omega}_\tau^i, \hat{H}_\tau^i, \Theta \right] | \theta_{NN}^*) - \hat{y}_\tau^{(i)} \right)^2$$

- Through back-propagation, the surrogate provides with the gradient of φ(·) for “free”

MOTIVATING EXAMPLE

- The Bates model
  - An extension of the Black-Scholes-Merton model with stochastic volatility and jumps.
  - **4 observable states, 1 hidden state, and 8 parameters**.
  - Solution technique: Fourier inversion of conditional characteristic function.

- How to evaluate its out-of-sample hedging (or pricing) performance?
  - Re-estimate the model (e.g., daily) using the cross section of option prices.
  - Compute the hedge ratios based on the estimated model.
  - Compute the out-of-sample hedging errors.

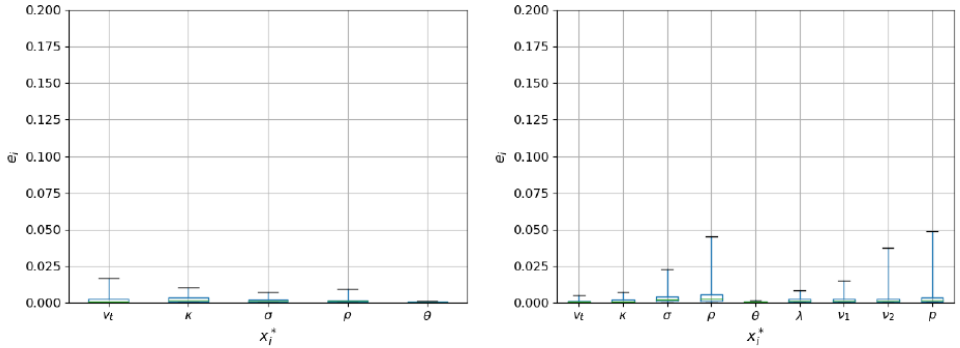
HOW LONG WILL IT TAKE?

- SPX: 4000 option prices on a typical day
- Modern FFT implementation (via Quantlib) vs. deep surrogate
- Single core vs. GPU

	FFT	Deep Surrogate	Deep Surrogate + GPU
pricing, 1-day	10s	0.6s	0.06s
estimation, 1-day	180s per iter	3.2s per iter	0.3s per iter
estimation, 1-year	125h	2.2h	.2h

CONTROLLED ENVIRONMENT

- To test our surrogate we, simulate N=1000 options with same parameters and hidden states, but randomly selected maturity and moneyness;
- use surrogate to estimate the parameters and hidden state;
- measure performance using relative estimation error: $e_i = \frac{|x_i^{true} - x_i^*|}{|\bar{x}_i - x_i|}$



(a) HM (b) BDJM

PRO'S & CON'S

- Con's: Pay an upfront cost to solve the model and create the surrogate.
- Pro: Once trained, the deep surrogate:
  - is highly accurate
  - is cheaper to use by orders of magnitude, including its gradients, which help with estimation
  - makes efficient use of GPUs
  - is easy to share (no longer need to build our own pricing models)
  - is easy to build on (with more training data, different architectures)

Example of BDJM

Approximator with a Cartesian grid

- 10 points per dimension (low) $\rightarrow 10^{13}$ (curse of dimensionality).
- $\sim 10^6$GB of storage.

Deep surrogate

- Training sample: Uniformly sample points (10^9) to train surrogate.
  - ↪ Could further improve efficiency with (adaptive) sparse grid.
- Only need to store the network's weights (20MB!)

THANK YOU FOR YOUR ATTENTION