Machine Learning for Structural Estimation

Accounting and Economics Society 2020 Summer School

Victor Duarte
UIUC Gies College of
Business

July, 29th 2020

Goals

- Tools and techniques to make structural estimation
  - Easier
  - Faster
  - Feasible
- Part I: tools and techniques
  - High-level overview of machine learning
  - Focus on solving and estimating models, not on empirical applications
- Part II: how to use these tools to solve and estimate (discrete-time) dynamic models

Approach

- Hands-on approach: lots of code!
  - Brief overview of concept and/or model description
  - Jump to the code that illustrates the idea and/or solves the model
- All code in Python: go to QuantEcon for a great intro
- Code in my GitHub (or just click the embedded links)
- We will run the code in Google Colab

Machine Learning

- Represent and learn high-dimensional nonlinear functions
  - Recognize a face on a picture
  - Play the game of Go (Link to the documentary)
- Can machine learning be a useful tool for your research?
  - Yes, if you work with functions!
- This lecture:
  - Deep Learning
  - Policy functions and moment functions

Why should you learn this?

- Machine learning has revolutionized many fields of research
  - E.g. image recognition, machine translation, intertemporal optimization
- Positive feedback between software, hardware, and methods
  - Better hardware makes adoption of new tech more appealing to researchers
  - New technology makes new methods possible
  - Larger pool of software developers leads to better software
  - Larger pool of customers provides incentives for hardware development
- Economics and finance can benefit from tapping into these resources

Outline

1. Software and Hardware
2. Stochastic Optimization
3. Neural Networks
4. Intertemporal Optimization
5. Moment Networks
6. Structural Estimation

Software and Hardware

Software

- Machine learning software: **TensorFlow (Google), PyTorch (Facebook)**
  - User-friendly, high-performance software suitable for AI applications
  - Highly scalable: run same code on laptop and on cloud
  - Front end is Python, back end is C++, CUDA or XLA
  - Open source
  - Support for differentiable programming
    - Can compute $\nabla_X Y$ where $X, Y =$ anything in your model
    - Used in ML to get derivative of network with respect to network parameters
- Does away with trade off between performance and development (Duarte et al., 2020)

Hardware

- Graphics Processing Units (GPU)
  - Developed by NVIDIA to accelerate the rendering of video games (1999)
  - *Tapping the Supercomputer Under Your Desk: Solving Dynamic Equilibrium Models with Graphics Processors* (JEDC, 2011)

  "Suffice it to say that, as the GPU computing technology matures, these details will become irrelevant for the average user (as they are nowadays for CPUs."

- Tensor Processing Units (TPU)

Example: Arellano (AER 2008)

- A sovereign default model solved with value iteration
- Reference: QuantEcon
- Experiment: solve it with specialized ML software and hardware and compare it to familiar programming languages.
- Full TensorFlow code here

Results: Performance comparison (Duarte et al., 2020)

		Grid size (for bond holdings)							
		151	351	551	751	951	1151	1351	1551
Hardware	**Software**	Average run time of one iteration							
Laptop	C++	37	178	578	1010	1674	2335	3158	4161
	Julia	169	725	1826	3370	5310	9188	28741	58269
	Matlab	91	318	792	1546	5215	23862	60801	98609
	Python/Numpy	133	667	1662	3068	11646	31228	51633	124027
	PyTorch	73	371	900	1648	2672	3969	5396	7445
	R	430	2237	5379	9917	15993	23726	33416	45791
	TensorFlow	20	117	291	533	859	1249	1752	2306
Desktop with GPU	PyTorch	0.32	1.40	3.63	7.37	12.02	16.14	19.37	20.67
	TensorFlow	0.42	0.79	1.25	1.75	2.50	3.44	4.79	6.18
Google Colab (GPU)	PyTorch	0.48	2.41	5.90	11.18	17.90	26.80	35.75	49.86
	TensorFlow	0.74	1.28	1.98	2.95	4.07	5.70	7.97	10.15
Google Colab (TPU)	TensorFlow	3.27	4.21	5.44	4.59	5.09	5.19	6.53	7.36
		Average Bellman error							
		-4.916	-4.919	-4.922	-4.923	-4.925	-4.923	-4.928	-4.932

This table shows the execution time (in milliseconds) of one iteration of the solution algorithm for the sovereign default model in Arellano (2008).

Comparing the code: Python/Numpy vs. PyTorch

```python
import numpy as np
logy_grid = np.loadtxt('logy_grid.txt')
Py = np.loadtxt('P.txt')

def main(nB=351, repeats=500):
    β, γ, r, θ = .953, 2., 0.017, 0.282
    ny = len(logy_grid)
    Bgrid = np.linspace(-.45, .45, nB)
    ygrid = np.exp(logy_grid)
    def_y = np.minimum(0.969 * np.mean(ygrid), ygrid)
    Vd = np.zeros([ny, 1])
    Vc = np.zeros((ny, nB))
    V = np.zeros((ny, nB))
    Q = np.ones((ny, nB)) * .95
    y = np.reshape(ygrid, [-1, 1, 1])
    B = np.reshape(Bgrid, [1, -1, 1])
    Bnext = np.reshape(Bgrid, [1, 1, -1])

    def u(c):
        return c**(1 - γ) / (1 - γ)

    def iterate(V, Vc, Vd, Q):
        EV = np.dot(Py, V)
        EVd = np.dot(Py, Vd)
        EVc = np.dot(Py, Vc)
        Vd_target = u(def_y) + β * (θ * EVc[:, nB // 2] + (1 - θ) * EVd[:, 0])
        Vd_target = np.reshape(Vd_target, [-1, 1])
        Qnext = np.reshape(Q, [ny, 1, nB])
        c = np.maximum(y - Qnext * Bnext + B, 1e-14)
        EV = np.expand_dims(EV, axis=1)
        m = u(c) + β * EV
        Vc_target = np.max(m, axis=2)
        default_states = Vd > Vc
        default_prob = np.dot(Py, default_states)
        Q_target = (1 - default_prob) / (1 + r)
        V_target = np.maximum(Vc, Vd)
        return V_target, Vc_target, Vd_target, Q_target


    for iteration in range(repeats):
        V, Vc, Vd, Q = iterate(V, Vc, Vd, Q)
    return V, Vc, Vd, Q
```

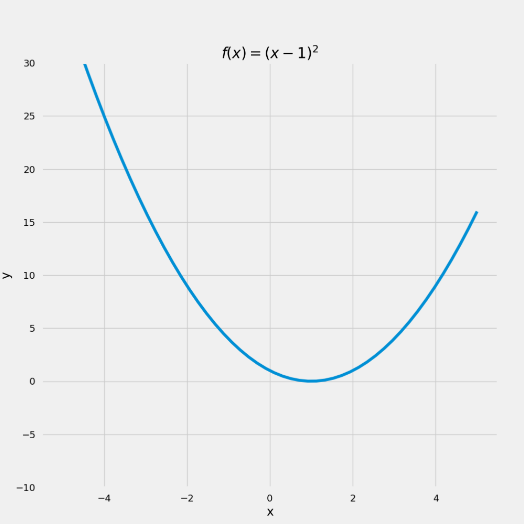
```python
import torch, numpy as np
logy_grid = torch.tensor(np.loadtxt('logy_grid.txt', dtype=np.float32))
Py = torch.tensor(np.loadtxt('P.txt', dtype=np.float32))

def main(nB=351, repeats=500):
    β, γ, r, θ = .953, 2., 0.017, 0.282
    ny = len(logy_grid)
    Bgrid = torch.linspace(-.45, .45, nB)
    ygrid = torch.exp(logy_grid)
    def_y = torch.min(torch.mean(ygrid), ygrid)
    Vd = torch.zeros([ny, 1])
    Vc = torch.zeros((ny, nB))
    V = torch.zeros((ny, nB))
    Q = torch.ones((ny, nB)) * .95
    y = torch.reshape(ygrid, [-1, 1, 1])
    B = torch.reshape(Bgrid, [1, -1, 1])
    Bnext = torch.reshape(Bgrid, [1, 1, -1])

    def u(c):
        return c**(1 - γ) / (1 - γ)

    def iterate(V, Vc, Vd, Q):
        EV = torch.matmul(Py, V)
        EVd = torch.matmul(Py, Vd)
        EVc = torch.matmul(Py, Vc)
        Vd_target = u(def_y) + β * (θ * EVc[:, nB // 2] + (1 - θ) * EVd[:, 0])
        Vd_target = torch.reshape(Vd_target, [-1, 1])
        Qnext = torch.reshape(Q, [ny, 1, nB])
        c = torch.relu(y - Qnext * Bnext + B)
        EV = torch.reshape(EV, [ny, 1, nB])
        m = u(c) + β * EV
        Vc_target = torch.max(m, dim=2, out=None)[0]
        default_states = (Vd > Vc).float()
        default_prob = torch.matmul(Py, default_states)
        Q_target = (1 - default_prob) / (1 + r)
        V_target = torch.max(Vc, Vd)
        return V_target, Vc_target, Vd_target, Q_target

    iterate = torch.jit.trace(iterate, (V, Vc, Vd, Q))  # Jit compilation
    for iteration in range(repeats):
        V, Vc, Vd, Q = iterate(V, Vc, Vd, Q)
    return V, Vc, Vd, Q
```

Stochastic Optimization

Gradient Descent with 1 Variable

- Problem:
  - Given a function $f : \mathbb{R} \to \mathbb{R}$
  - Find $\min_x f(x)$
  - Assuming you have only *local* information: $f'(x)$

Gradient Descent with 1 Variable



- Start with an arbitrary initial guess, for instance $x_0 = 4$
- $\Delta y \approx f'(x_0)\Delta x$
- If $f'(x_0) > 0$, then $\Delta x < 0$
- If $f'(x_0) < 0$, then $\Delta x > 0$
- Gradient descent:

$$\Delta x \propto -f'(x_0)$$

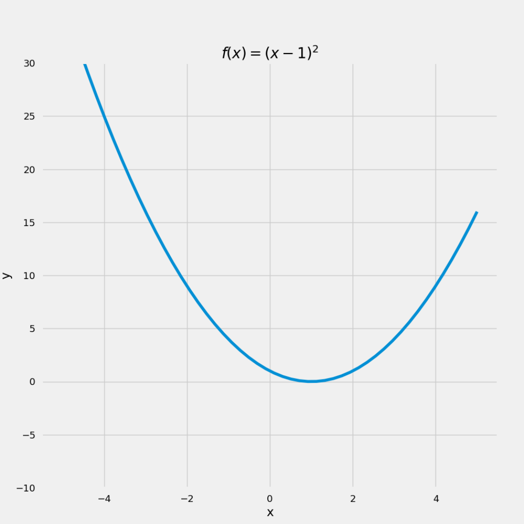
$$\Delta x = -\alpha f'(x_0)$$

1: Univariate Gradient Descent

$$x_1 = x_0 - \alpha f'(x_0)$$

- α is called the *learning rate*

Gradient Descent with 1 Variable



2: Univariate Gradient Descent

$$x_1 = x_0 - \alpha f'(x_0)$$

x	f(x)	f'(x)
4	9	6
3.4	5.76	4.8
2.92	3.686	3.84
2.536	2.359	3.072
2.229	1.51	2.458
1.983	0.966	1.966
⋮	⋮	⋮
1	0	0

Code

Multivariate Gradient Descent

- Problem:
  - Given a function $f : \mathbb{R}^n \to \mathbb{R}$
  - Find $\min_x f(x)$
  - Assuming you have only *local* information: $\nabla_x f(x)$

Multivariate Gradient Descent

- Problem:
  - Given a function $f : \mathbb{R}^n \to \mathbb{R}$
  - Find $\min_x f(x)$
  - Assuming you have only *local* information: $\nabla_x f(x)$

4: Gradient Descent

$$x_1 = x_0 - \alpha \nabla_x f(x)$$

Multivariate Gradient Descent

- Justification:
  - The gradient points in the direction of steepest ascent.
- Proof:

$$f(\overrightarrow{x_0} + \overrightarrow{h}\epsilon) \approx f(x_0) + \langle \nabla f(x_0), \overrightarrow{h} \rangle \epsilon$$

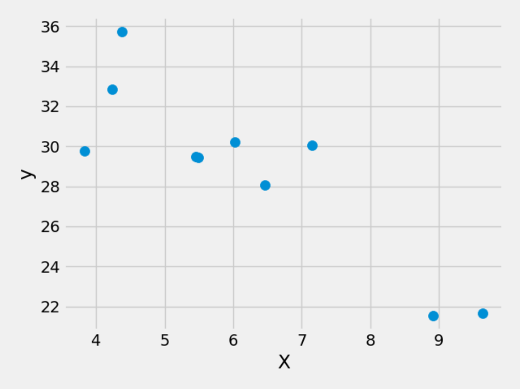
  - where $\overrightarrow{h}$ is a unit vector ($||h|| = 1$)
  - By the Cauchy-Schwarz inequality, the right-hand side of the equation above is maximized for

$$\overrightarrow{h} = \frac{\nabla f(x_0)}{||\nabla f(x_0)||}$$

Linear Regression (1)

- Given data of pairs $\{X_i, y_i\}_{i=0}^{N}$
- What is the best linear function that fits the data?
- $f(x; \Theta) = \theta_0 + \theta_1 x$

Example:



Linear Regression (2)

- Mean squared loss function:

$$L(\Theta) = \frac{1}{2N} \sum_{i=0}^{N} (f(x_i; \Theta) - y_i)^2$$

- Gradient of the loss function:

$$\nabla_{\Theta} L(\Theta) = \frac{1}{N} \sum_{i=0}^{N} (f(x_i; \Theta) - y_i) \nabla f(x_i; \Theta)$$

Linear Regression (3)

- Notice that $\nabla_{\Theta} f(x_i, ; \Theta) = [1 \ x_i]^T$
- Applying the gradient descent update we get:

$$\Delta\theta_0 = \frac{1}{N}\sum_{i=0}^{N}(f(x_i; \Theta) - y_i)$$

$$\Delta\theta_1 = \frac{1}{N}\sum_{i=0}^{N}(f(x_i; \Theta) - y_i)x_i$$

Code

Stochastic Gradient Descent - Large-scale ML

- Training data: $\{(x_1, y_1), ..., (x_n, y_n)\}$, $x_i \in \mathbb{R}^d$
- large-scale ML: n and d are large:
  - $d =$ number of features
  - $n =$ number of samples

$$L(\Theta) = \frac{1}{n} \sum_{i=0}^{n} l(x_i, y_i; \Theta)$$

$$\Theta_1 = \Theta_0 - \alpha \frac{1}{n} \sum_{i=0}^{n} \nabla_\Theta l(x_i, y_i; \Theta)$$

- Drawbacks?

Stochastic Gradient Descent - Large-scale ML

- At iteration k, randomly pick an integer

$$i(k) \in \{1, 2, ..., n\}$$

- Perform the update:

$$\Theta_1 = \Theta_0 - \alpha \nabla_\Theta l(x_{i(k)}, y_{i(k)}; \Theta)$$

- Does this make sense?

Stochastic Gradient Descent - Large-scale ML

- At iteration k, randomly pick an integer

$$i(k) \in \{1, 2, ..., n\}$$

- Perform the update:

$$\Theta_1 = \Theta_0 - \alpha \nabla_\Theta l(x_{i(k)}, y_{i(k)}; \Theta)$$

- Does this make sense?
  - Yes! (Robbins and Monro (1951) - “A Stochastic Approximation Method”)

Stochastic Gradient Descent - Large-scale ML

- Visualization
  - Gradient Descent
  - Stochastic Gradient Descent

Stochastic Gradient Descent - Mini-batch

- Use a *mini-batch* of stochastic gradients

$$\Theta_1 = \Theta_0 - \alpha \frac{1}{|I|} \sum_{j \in I_k} \nabla_\Theta l(x_j, y_j; \Theta)$$

- Each iteration uses $|I_k|$ stochastic gradients
- Useful in parallel setttings (Multi-core CPUs, GPUs, TPUs)

Stochastic Gradient Descent - Advanced Optimization

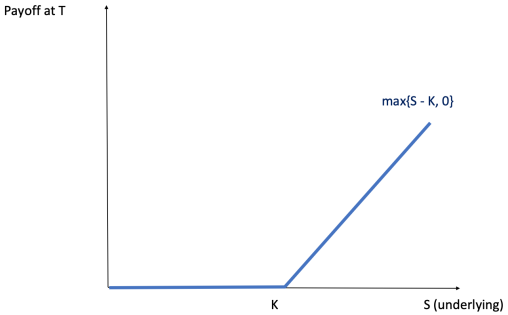
- Momentum
- RMSProp
- Adam
- And many more ...

Neural Networks

Neural Networks and Options

- Payoff of a call option:

$$C(S, K) = \max\{S - K, 0\}$$



General Combinations of Options (1)

- Payoff of a portfolio of N call options with strike prices $\{K_1, K_2, \ldots, K_N\}$:

$$V(S) = \sum_{i=1}^{N} \Theta_i \cdot \max\{S - K_i,\ \ 0\}$$

General Combinations of Options (2)

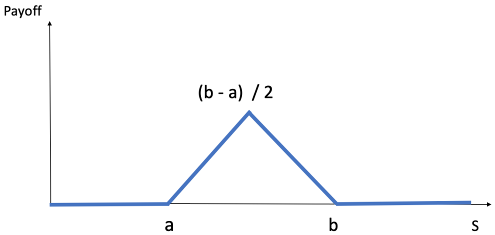
- Suppose you are certain that the price of a particular stock, S_T, will be in the inverval:

$$S_T \in [a, b]$$

  Consider the following strategy:

  - Purchase 1 call with an exerise price a
  - Sell 2 calls with exerise prices $(a+b)/2$
  - Purchase 1 calls with exerise prices of b

Code



General Combinations of Options (3)

- Options are extremely flexible securities that allow market participants to focus on very particular outcomes of the underlying securities

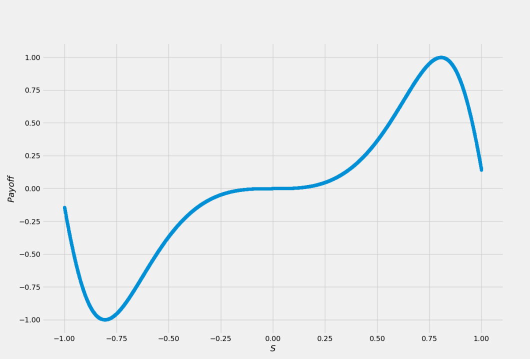
General Combinations of Options (3)

- Options are extremely flexible securities that allow market participants to focus on very particular outcomes of the underlying securities

- What about more complex payoffs?

General Combinations of Options (3)

- Options are extremely flexible securities that allow market participants to focus on very particular outcomes of the underlying securities
- What about more complex payoffs?



Code

Option Spanning Theorem

Theorem (Steve Ross (1976))

Any contract can be replicated or spanned by a suitable combination of options.

- Payoff of a portfolio of call options:

$$V(S) = \sum_{i=1}^{N} \Theta_i \cdot \max\{S - K_i, \ 0\}$$

Option Spanning Theorem

Theorem (Steve Ross (1976))

Any contract can be replicated or spanned by a suitable combination of options.

- Payoff of a portfolio of call options:

$$V(S) = \sum_{i=1}^{N} \Theta_i \cdot \max\{S - K_i,\ \ 0\}$$

- A *single-layer*, *feedforward* neural network with *relu activation* that takes as input a single feature S is:

$$f(S) = \sum_{i=1}^{N} \Theta_i \cdot \max\{W_i S - K_i,\ \ 0\}$$

Option Spanning Theorem

Theorem (Steve Ross (1976))

Any contract can be replicated or spanned by a suitable combination of options.

- Payoff of a portfolio of call options:

$$V(S) = \sum_{i=1}^{N} \Theta_i \cdot \max\{S - K_i,\ \ 0\}$$

- A *single-layer*, *feedforward* neural network with *relu activation* that takes as input a single feature S is:

$$f(S) = \sum_{i=1}^{N} \Theta_i \cdot \max\{W_i S - K_i,\ \ 0\}$$

- A *single-layer*, *feedforward* neural network with *relu activation* that takes as input k *features* $\mathbf{x} = x_1, x_2, \ldots, x_k$ is:

$$f(\mathbf{x}) = \sum_{i=1}^{N} \Theta_i \cdot \max\{W_i^T \mathbf{x} - K_i,\ \ 0\}$$

Feed-forward Neural Network Graph Representation

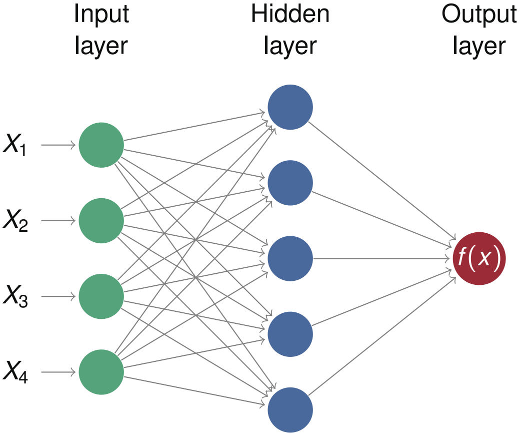


Figure: Architecture of a Single Layer Network

- Each blue circle represents a *hidden unit, or neuron* (or a call option in our analogy)
- The parameters of the network are called *weights*

Feed-forward Neural Network Graph Representation

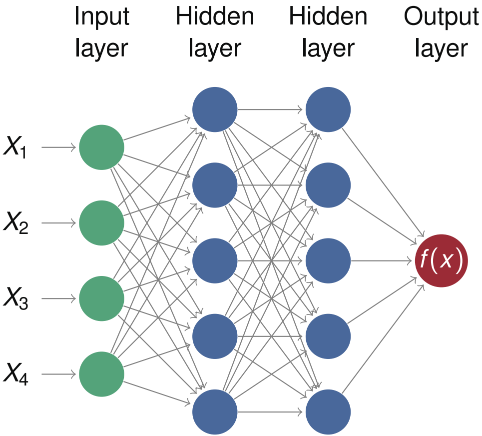
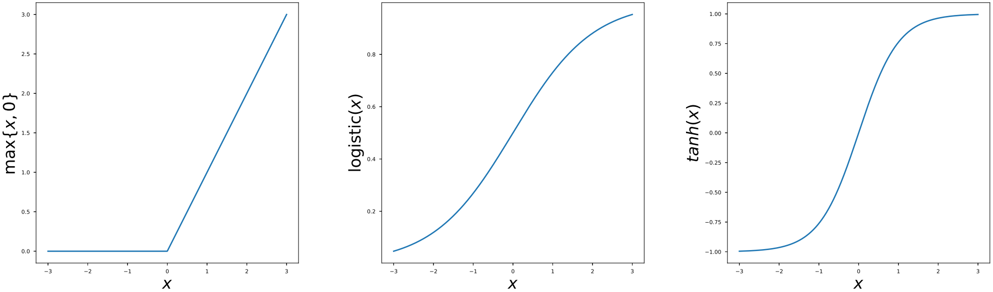


Figure: Architecture of a Deep Neural Network

- Neural networks with multiple hidden layers are called *deep neural networks*
- Each hidden unit consists of a linear combination of the previous units followed by a nonlinear *activation*

Other activation functions

Figure: Activation functions



(a) (b) (c)

Universal Approximation Theorem

Theorem (Cybenko, G. (1989))

A single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathbb{R}^n$*.*

Automatic Differentiation

- To train a neural network, we need the gradient of a loss function w.r.t. *all* network parameters
- Autodiff (more specifically backpropagation) computes $\nabla_\Theta L(\Theta)$ with the same cost of computing $L(\Theta)$
- Gradients are exact (up to machine precision)
- $grad = tf.gradients(L, \Theta)$
- Further reference: Backpropagation

A simple example

- Goal: Train a neural network to learn the function

$$f : [0, 1] \to \mathbb{R}, \ f(x) = x^2$$

- Data (x, x^2) with $x \in [0, 1]$
- We will use neural 2-layer network (with random initial weights).
- Minimize the mean squared error loss using stochastic gradient descent
- We will use a batch size of 10000 observations

Code

Another example: Fashion MNIST

Reference: TF tutorial

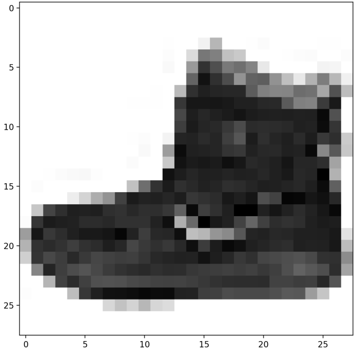
- 60,000 gray-scale images
- 10 classes:
    0. T-shirt/top
    1. Trouser
    2. Pullover
    3. Dress
    4. Coat
    5. Sandal
    6. Shirt
    7. Sneaker
    8. Bag
    9. Ankle boot
- Goal: build a neural network that learns to classify these images

Another example: Fashion MNIST

- Sample images

Another example: Fashion MNIST

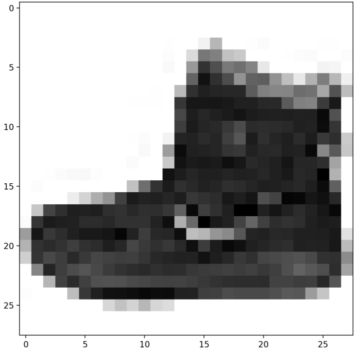
- Each image is a 28×28 matrix



- Each of the $784 (28 \times 28)$ pixels is a real number between 0 (white) and 1 (black)

Another example: Fashion MNIST

- Each image is a 28×28 matrix



- Each of the $784 (28 \times 28)$ pixels is a real number between 0 (white) and 1 (black)
- Each image is vector of 784 features

Another example: Fashion MNIST

- 10 classes:
  0. T-shirt/top
  1. Trouser
  2. Pullover
  3. Dress
  4. Coat
  5. Sandal
  6. Shirt
  7. Sneaker
  8. Bag
  9. Ankle boot

- One-hot encoding:
  - a T-shirt is represented by the vector $[1, 0, 0, \ldots, 0]$
  - an Ankle boot is represented by the vector $[0, 0, 0, \ldots, 1]$

Another example: Fashion MNIST

- We will design a neural net that
  - takes as inputs a vector of 784 features
  - outputs a vector of 10 numbers: the conditional probabilities of the image belonging to each of the 10 classes.

Code

Intertemporal Optimization

Direct Policy Search (1)

Duarte, Fonseca, Goodman and Parker (Work in progress)

- Expected lifetime utility is given by:

$$\mathbb{E}\left[\sum_{t=0}^{T} \beta^t u(C_t(s_t))\right]$$

- We parametrize policy functions using neural networks

$$C_t(s_t) \equiv C(s_t; \Theta_t)$$

- Loss function:

$$\mathcal{L}(\Theta) = -\mathbb{E}\left[\sum_{t=0}^{T} \beta^t u(C(s_t; \Theta_t))\right]$$

Direct Policy Search (2)

Duarte, Fonseca, Goodman and Parker (Work in progress)

- Gradient descent update

$$\Delta\Theta = -\alpha\nabla_{\Theta}\mathcal{L}(\Theta)$$

- Plugging in the loss function

$$\Delta\Theta = -\alpha\nabla_{\Theta}\left(-\mathbb{E}\left[\sum_{t=0}^{T}\beta^{t}u(C(s_t;\Theta_t))\right]\right)$$

- Swapping the order of $\mathbb{E}$ and ∇

$$\Delta\Theta = \alpha\mathbb{E}\left(\nabla_{\Theta}\left[\sum_{t=0}^{T}\beta^{t}u(C(s_t;\Theta_t))\right]\right)$$

- Stochastic gradient descent update

$$\Delta\Theta \approx \alpha\frac{1}{N}\sum_{i=1}^{N}\nabla_{\Theta}\left(\sum_{t=0}^{T}\beta^{t}u(C(s_t;\Theta_t))\right)$$

- This gradient is known as pathwise gradient estimator (Survey)

Direct Policy Search (3)

Duarte, Fonseca, Goodman and Parker (Work in progress)

- In code:

```
@tf.function
def train_step():
    optimizer.minimize(simulate, θ)
```

Lifecycle Model Example (Fernandez-Villaverde and Valencia, 2018)

We consider the following model for our experiments. We have an economy populated by a continuum of individuals of measure one who live for T periods. Individuals supply inelastically one unit of labor each period and receive labor income according to a market-determined wage, w, and an idiosyncratic and uninsurable productivity shock e. This shock follows an age-independent Markov chain, with the probability of moving from shock e_j to e_k given by $\mathbb{P}(e_k|e_j)$. Individuals have access to one-period risk-free bonds, x, with return given by a market-determined net interest rate, r.

Given the age, t, an exogenous productivity shock, e, and savings from last period, x, the individual chooses the optimal amount of consumption, c, and savings to carry for next period, x'. The problem of the household during periods $t \in \{1, \ldots, T-1\}$ is described by the following Bellman equation:

$$
\begin{aligned}
V(t, x, e) = \max_{\{c \geq 0, x'\}} \quad & u(c) + \beta \mathbb{E} V(t+1, x', e') \\
\text{s.t. } & c + x' = (1+r)x + ew \\
& e' \sim \mathbb{P}(e'|e),
\end{aligned}
$$

for some initial savings and productivity shock.[10] The problem of the household in period T is just to consume all resources, yielding a terminal condition for the value function:

$$V(T, x, e) = u((1+r)x + ew).$$

Code

Including parameters as inputs of the neural networks

- Standard solution
  - Solve a dynamic programming problem for each set of parameters β_i
  - policy = policy(states)
- Gradient-Based Structural Estimation (Duarte, 2018)
  - Take the parameters as inputs for the policies
  - policy = policy(states, β)
  - Solve the dynamic programming problem once

Code

Moment Networks

Problem

- General problem:
  - vector of data moments $\hat{g}$
  - vector of model-implied moments $g(\beta) = \mathbb{E}[Y|\beta]$

$$\beta^* = argmin||\hat{g} - g(\beta)||_W^2$$

- We cannot (yet) use gradient-based optimization
  - We don't have a mapping between model parameters and model-implied moments

Solution (1)

- Claim:
  - We can learn the mapping $g(\beta) = \mathbb{E}[Y|\beta]$ by observing enough data $\{\beta_i, Y_i\}$
- Proof:
  - Supervised learning (nonlinear regression) solves the problem:

$$g(\beta) = argmin_f \mathbb{E}[(Y - f(\beta))^2]$$

  - The conditional expectation also solves the MSE problem:

$$g(\beta) = \mathbb{E}[Y|\beta]$$

Solution (2)

- $g(\beta)$ is a function
  - Approximated by a neural network
  - Alleviates the curse of dimensionality
- $g(\beta)$ is as good as an analytical formula
  1. Evaluating the moments takes a few milliseconds
  2. Differentiable
  3. Can use the gradient-based methods to estimate the parameters

Algorithm (1)

1. Draw a (large) sample of model parameters β_i (uniformly for now)
2. Simulate the model for each parameter and record the vector of observations Y_i

Algorithm (1)

1. Draw a (large) sample of model parameters β_i (uniformly for now)

2. Simulate the model for each parameter and record the vector of observations Y_i

	parameters (β_i)			observations (Y_i)		
observation	γ	ψ	β	L	C	C^2
1	2.1	1.5	.97	1.3	.1	3.9
2	3.2	1.8	.96	1.1	.9	2.8
3	2.7	2.0	.99	1.1	.3	4.3
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Algorithm (2)

3. Feed the data to machine learning software

4. Result: $g(\beta) = \begin{pmatrix} \mathbb{E}[L|\beta] \\ \mathbb{E}[C|\beta] \\ \mathbb{E}[C^2|\beta] \end{pmatrix}$

5. Minimize objective function with gradient-based methods

$$\beta^* = argmin||\hat{g} - g(\beta)||_W^2$$

Intuition

$\beta_1 = 1$

Intuition

$\beta_1 = 1$ tons of computations

Intuition

$\beta_1 = 1$ tons of computations $g(\beta_1) = \mathbb{E}[Y|\beta_1] = 2$

Intuition

$\beta_1 = 1$ tons of computations $g(\beta_1) = \mathbb{E}[Y|\beta_1] = 2$

$\beta_2 = 2$ tons of computations $g(\beta_2) = \mathbb{E}[Y|\beta_2] = 4$

Intuition

$\beta_1 = 1$ tons of computations $g(\beta_1) = \mathbb{E}[Y|\beta_1] = 2$

$\beta_2 = 2$ tons of computations $g(\beta_2) = \mathbb{E}[Y|\beta_2] = 4$

$\beta_3 = 1.5$

Intuition

$\beta_1 = 1$	tons of computations	$g(\beta_1) = \mathbb{E}[Y \mid \beta_1] = 2$
$\beta_2 = 2$	tons of computations	$g(\beta_2) = \mathbb{E}[Y \mid \beta_2] = 4$
$\beta_3 = 1.5$	"*probably*"	$g(\beta_3) = \mathbb{E}[Y \mid \beta_3] = 3$

Intuition

$\beta_1 = 1$ some computations $Y|\beta_1 = 2 + \varepsilon$

$\beta_2 = 2$ some computations $Y|\beta_2 = 4 + \varepsilon$

$\beta_3 = 1.5$ *"probably"* $g(\beta_3) = \mathbb{E}[Y|\beta_3] = 3$

Moment Networks - Example

- Let $g(a)$ be a function that takes as input the parameter a and returns an expectation:

$$g(a) = \mathbb{E}[cos(a + \varepsilon)]$$

- Suppose we want to find the parameter a s.t. $g(a) = 0.1$
- Strategy:
  - Construct a network to approximate g
  - Choose a to minimize $(g(a) - 0.1)^2$

Code

Structural Estimation

Using moment networks with intertemporal optimization

- Back to our lifecycle example
- Suppose we want to choose risk aversion γ to match average consumption in the terminal date ($\mathbb{E}[C_9] = 5.87$)
- The solution to this problem is $\gamma = 2$
- Strategy:
  - Include parameters as inputs to the policy networks (state variables)
  - Construct a network to approximate $g(\gamma) = \mathbb{E}[C_9|\gamma]$
  - Minimize the distance $(g(\gamma) - 5.87)^2$

Code