Deep learning and reinforcement learning

Jesús Fernández-Villaverde[1] and Galo Nuño[2]

October 15, 2021

[1]University of Pennsylvania

[2]Banco de España

A short introduction

The problem

- Let us suppose we want to approximate an unknown function:

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

  where y is a scalar and $\mathbf{x} = \{x_1, x_2, ..., x_N\}$ a vector.

- We care about the case when N is large.

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

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

A neural network

- An artificial neural network (a.k.a. ANN or connectionist system) is an approximation to $f(\mathbf{x})$ built as a linear combination of M generalized linear models of $\mathbf{x}$ of the form:

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

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

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

- M is known as the width of the model.
- We can select θ such that $g^{NN}(\mathbf{x}; \theta)$ is as close to $f(\mathbf{x})$ as possible given some relevant metric (e.g., L^2 norm).
- This is known as "training" the network.

Comparison with other approximations

- Compare:

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

  with a standard projection:

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

  where ϕ_m is, for example, a Chebyshev polynomial.

- We exchange the rich parameterization of coefficients for the parsimony of basis functions.
- Later, we will explain why this is often a good idea.
- How we determine the coefficients will also be different, but this is somewhat less important.

Deep learning

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

$$y \cong g^{DL}(\mathbf{x}; \theta) = g^{NN(1)}\left(g^{NN(2)}\left(...; \theta^{(2)}\right); \theta^{(1)}\right)$$

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

- Sometimes known as deep feedforward neural networks or multilayer perceptrons.

- "Feedforward" comes from the fact that the composition of neural networks can be represented as a directed acyclic graph, which lacks feedback. We can have more general recurrent structures.

- J is known as the depth of the network. The case $J = 1$ is a standard neural network.

- As before, we can select θ such that $g^{DL}(\mathbf{x}; \theta)$ approximates a target function $f(\mathbf{x})$ as closely as possible under some relevant metric.

Why are neural networks a good solution method in economics?

- From now on, I will refer to neural networks as including both single and multilayer networks.
- With suitable choices of activation functions, neural networks can efficiently approximate extremely complex functions.
- In particular, under certain (relatively weak) conditions:
    1. Neural networks are universal approximators.
    2. Neural networks break the "curse of dimensionality."
- Furthermore, neural networks are easy to code, stable, and scalable for multiprocressing.
- Thus, neural networks have considerable option value as solution methods in economics.

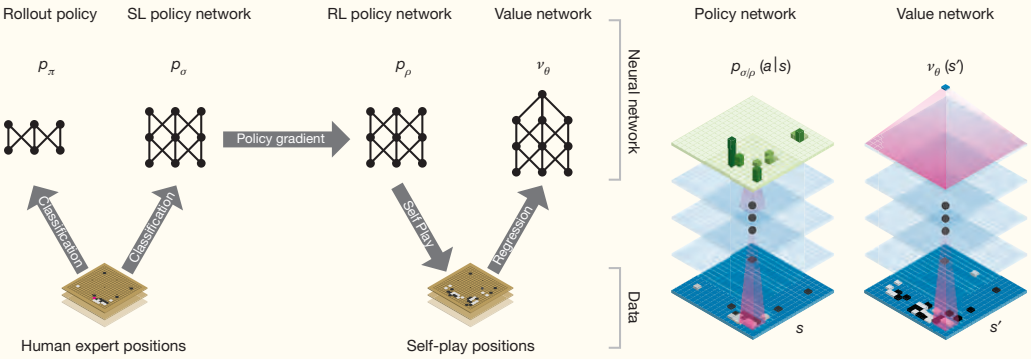
Current interest

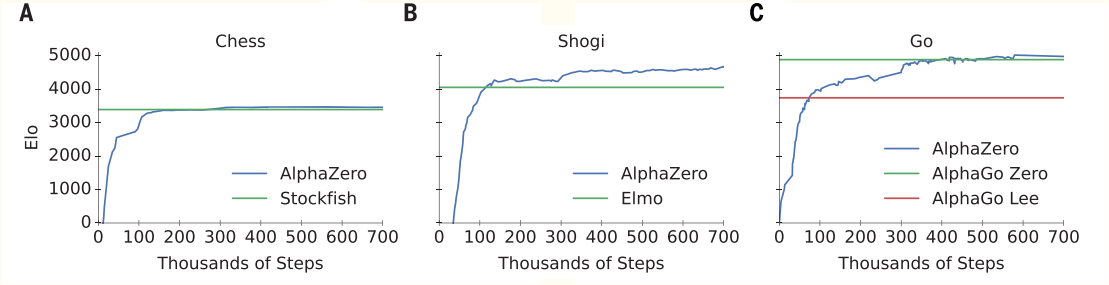
- Currently, neural networks are among the most active areas of research in computer science and applied math.
- While original idea goes back to the 1940s, neural networks were rediscovered in the second half of the 2000s.
- Why?
  1. Suddenly, the large computational and data requirements required to train the networks efficiently became available at a reasonable cost.
  2. New algorithms such as *back propagation* through gradient descent became popular.
- Some well-known successes and industrial applications.


ALPHAGO
01:58:30
LEE SEDOL
01:59:41
Mind
Match
AlphaGo
Lee Sedol

AlphaGo

- Big splash: AlphaGo vs. Lee Sedol in March 2016.
- Silver *et al.* (2018): now applied to chess, shogi, Go, and StarCraft II.
- Check also:
  1. https://deepmind.com/research/alphago/.
  2. https://www.alphagomovie.com/
  3. https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii
- Very different than Deep Blue against Kasparov.
- New and surprising strategies.
- However, you need to keep this accomplishment in perspective.


Rollout policy
SL policy network
RL policy network
Value network
p_π
p_σ
p_ρ
v_θ
Policy gradient
Classification
Classification
Self Play
Regression
Neural network
Data
Human expert positions
Self-play positions
Policy network
Value network
$p_{\sigma/\rho}(a|s)$
$v_\theta(s')$
s
s'


A
Chess
Elo
5000
4000
3000
2000
1000
0
AlphaZero
Stockfish
0 100 200 300 400 500 600 700
Thousands of Steps
B
Shogi
AlphaZero
Elmo
0 100 200 300 400 500 600 700
Thousands of Steps
C
Go
AlphaZero
AlphaGo Zero
AlphaGo Lee
0 100 200 300 400 500 600 700
Thousands of Steps

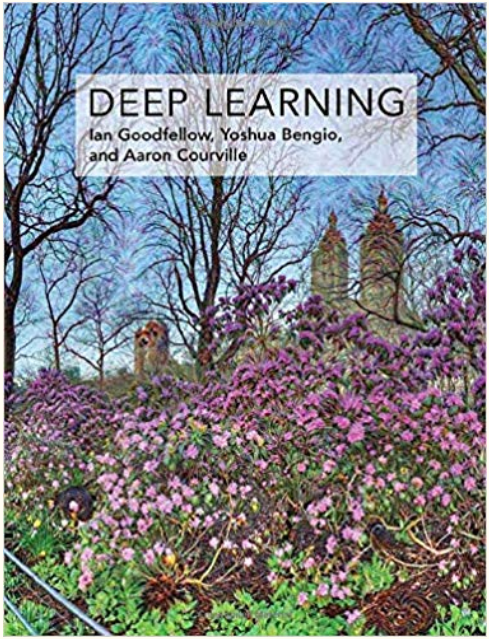
Further advantages

- Neural networks and deep learning often require less "inside knowledge" by experts on the area.
- Results can be highly counter-intuitive and yet, deliver excellent performance.
- Outstanding open source libraries: `Tensorflow`, `Pytorch`, `Flux`.
- More recently, development of dedicated hardware (TPUs, AI accelerators, FPGAs) are likely to maintain a hedge for the area.
- The width of an ecosystem is key for its long-run success.

TensorFlow

Limitations of neural networks and deep learning

- While neural networks and deep learning can work extremely well, there is no such a thing as a silver bullet.
- Clear and serious trade-offs in real-life applications.
- Rule-of-thumb in the industry is that one needs around 10^7 labeled observations to properly train a complex ANN with around 10^4 observations in each relevant group.
- Of course, sometimes "observations" are endogenous (we can simulate them), but if your goal is to forecast GDP next quarter, it is unlikely a neural network will beat an ARIMA(n,p,q) (at least only with macro variables).
- Issues of interpretation.


DEEP LEARNING
Ian Goodfellow, Yoshua Bengio,
and Aaron Courville

Digging deeper

More details on neural networks

- Non-linear functional approximation method.
- Much hype around them and over-emphasis of biological interpretation.
- We will follow a much sober formal treatment (which, in any case, agrees with state-of-art researchers approach).
- In particular, we will highlight connections with econometrics (e.g., NOLS, semiparametric regression, and sieves).
- We will start describing the simplest possible neural network.

A neuron

- N observables: x_1, x_2,...,x_N. We stack them in $\mathbf{x}$.
- Coefficients (or weights): θ_0 (a constant), θ_1, θ_2, ...,θ_N. We stack them in θ.
- We build a linear combination of observations:

$$z = \theta_0 + \sum_{n=1}^{N} \theta_n x_n$$

  Theoretically, we could build non-linear combinations, but unlikely to be a fruitful idea in general.

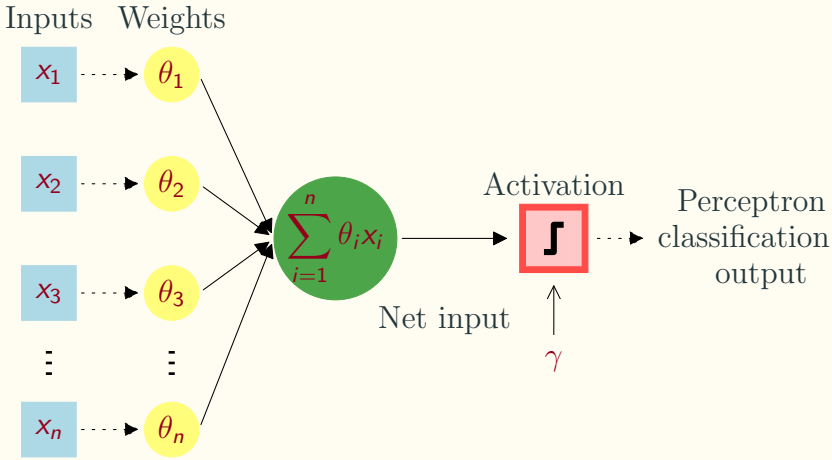
- We transform such linear combination with an activation function:

$$y = g(\mathbf{x}; \theta) = \phi(z)$$

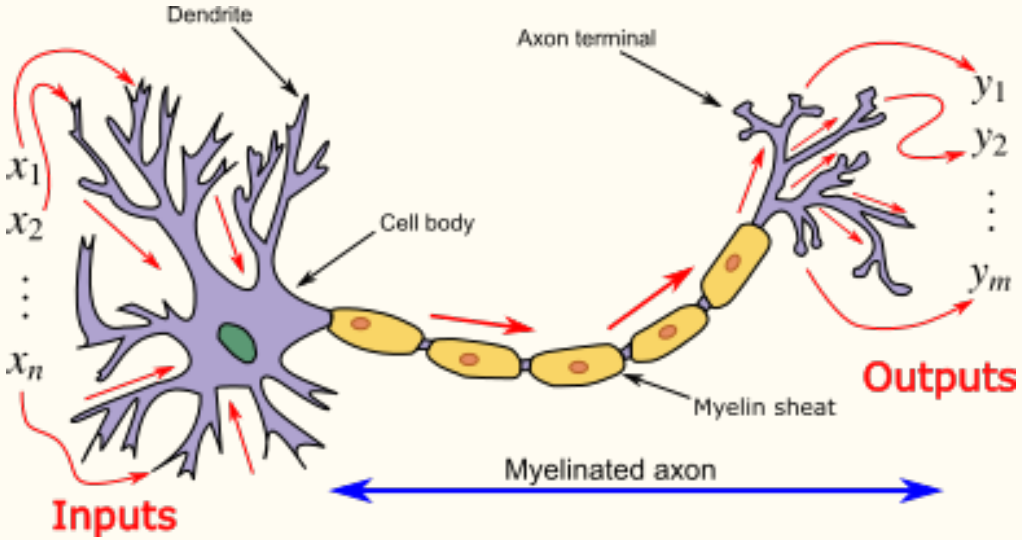
  The activation function might have some coefficients γ on its own.

- Why do we need an activation function?

Flow representation

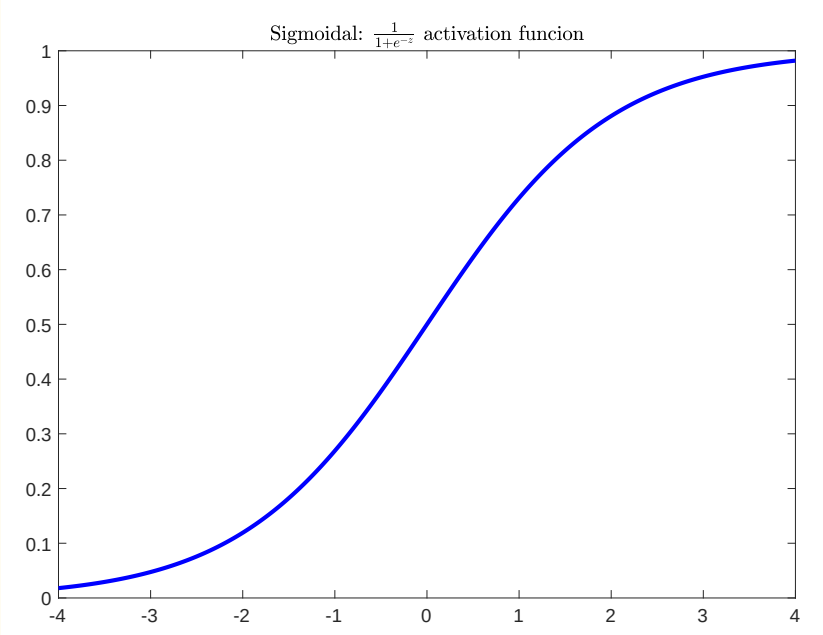


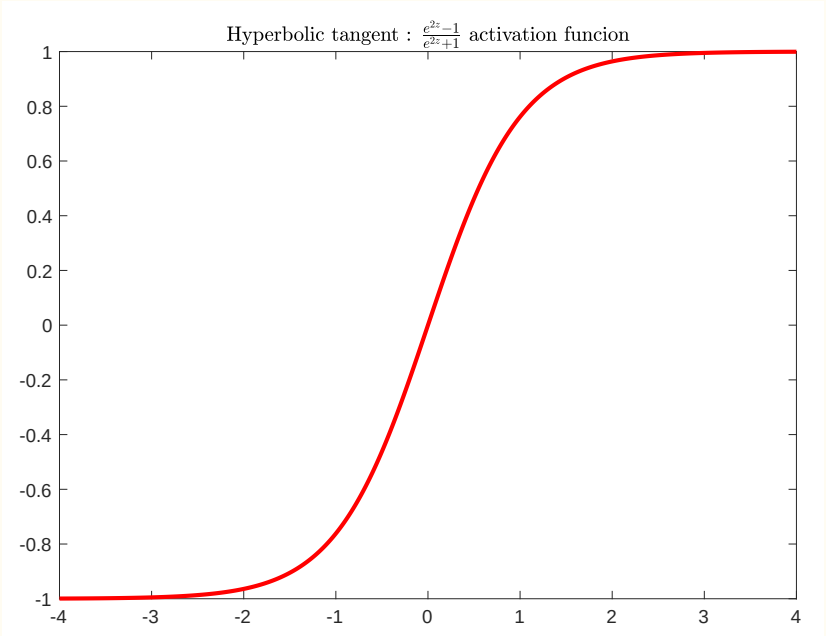
The biological analog



Activation functions I

- Traditionally:
  1. Identity function:
  $$\phi(z) = z$$
  Used in linear regression.
  2. A sigmoidal function:
  $$\phi(z) = \frac{1}{1 + e^{-z}}$$
  A particular limiting case as z grows quickly: step function.
  3. Hyperbolic tangent:
  $$\phi(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$


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


Hyperbolic tangent : $\frac{e^{2z}-1}{e^{2z}+1}$ activation funcion
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-4
-3
-2
-1
0
1
2
3
4

Activation functions II

- Some activation functions that have gained popularity recently:

1. Rectified linear unit (ReLU):

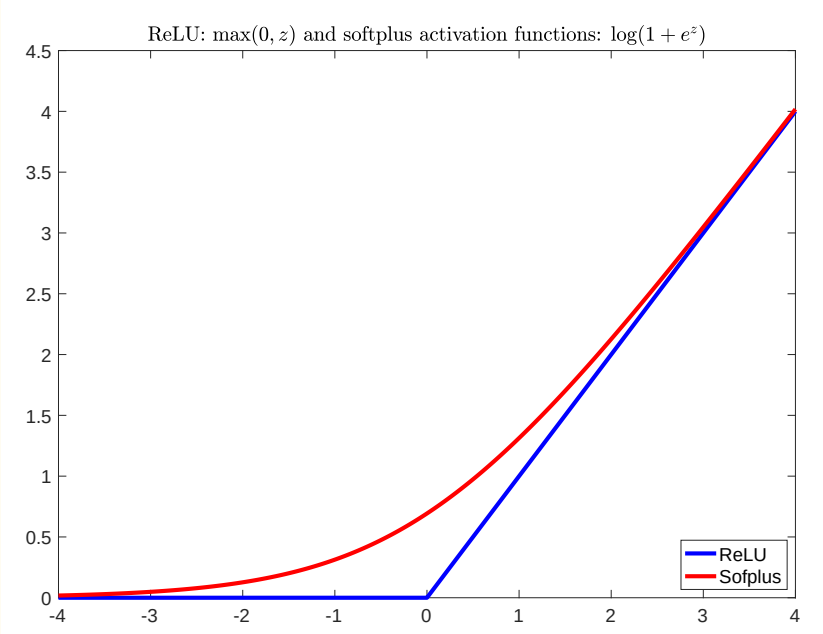
$$\phi(z) = max(0, z)$$

2. Parametric ReLU:

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

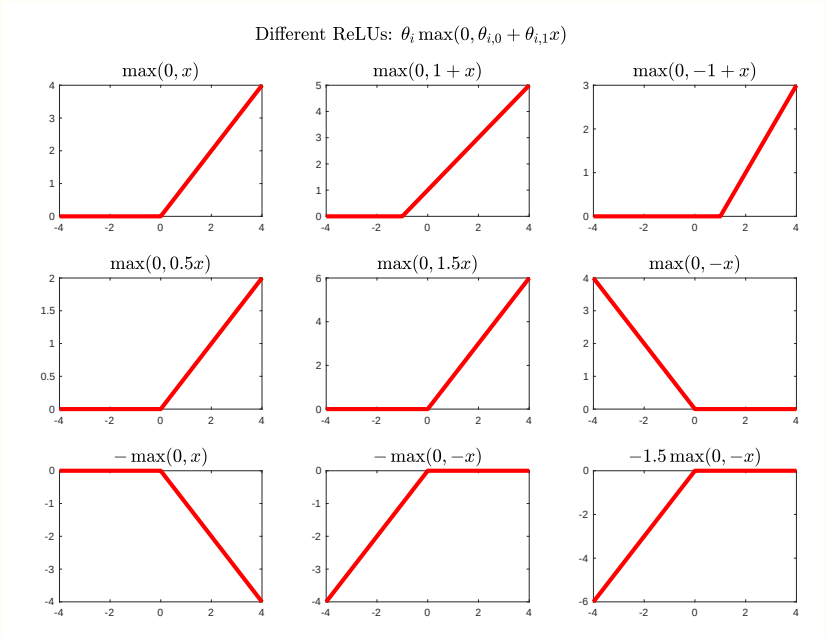
3. Softplus:

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


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

Interpretation

- θ_0 controls the activation threshold.
- The level of the θ_i's for $i > 0$ control the activation rate (the higher the θ_i's, the harder the activation).
- Some textbooks separate the activation threshold and scaling coefficients from θ as different coefficients in ϕ, but such separation moves notation farther away from standard econometrics.
- Potential identification problem between θ and more general activation functions with their own parameters.
- But in practice θ does not have a structural interpretation, so the identification problem is of secondary importance.
- As mentioned in the introduction, a neuron closely resembles a generalized linear model in econometrics.


Different ReLUs: $\theta_i \max(0, \theta_{i,0} + \theta_{i,1} x)$
$\max(0, x)$
$\max(0, 1 + x)$
$\max(0, -1 + x)$
$\max(0, 0.5x)$
$\max(0, 1.5x)$
$\max(0, -x)$
$-\max(0, x)$
$-\max(0, -x)$
$-1.5 \max(0, -x)$

Combining neurons into a neural network

- As before, we have N observables: x_1, x_2,...,x_N.
- Coefficients (or weights): $\theta_{0,m}$ (a constant), $\theta_{1,m}$, $\theta_{2,m}$, ...,$\theta_{N,m}$.
- We build M linear combinations of observations:

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

- We transform and add such linear combinations with an activation function:

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

- Also, quasi-linear structure in terms of vectors of observables and coefficients.
- This is known as a single layer network.

Two classic (yet remarkable) results I

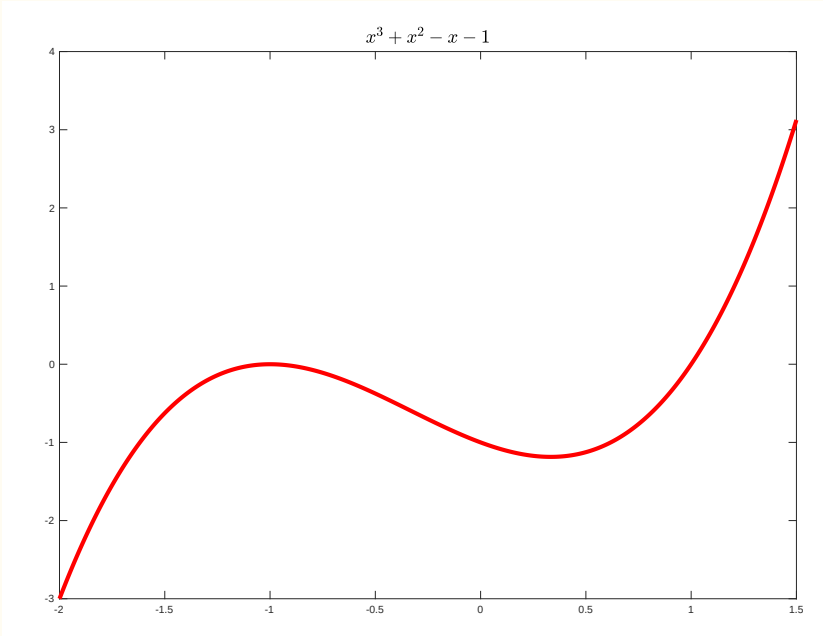
Borel measurable function

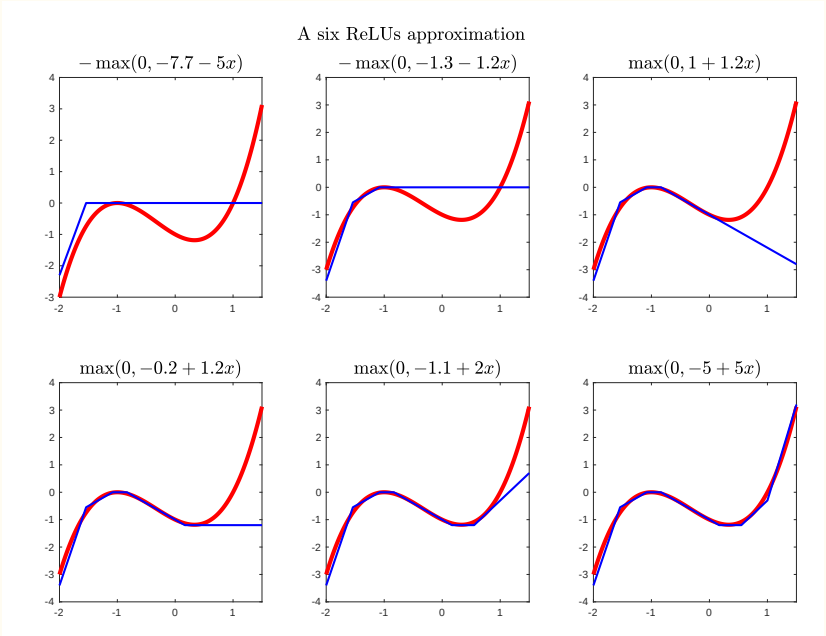
A map $f : X \to Y$ between two topological spaces is called Borel measurable if $f^{-1}(A)$ is a Borel set for any open set A on Y (the Borel sets are all the open sets built through the operations of countable union, countable intersection, and relative complement).

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

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

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


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


A six ReLUs approximation
$-\max(0, -7.7 - 5x)$
$-\max(0, -1.3 - 1.2x)$
$\max(0, 1 + 1.2x)$
$\max(0, -0.2 + 1.2x)$
$\max(0, -1.1 + 2x)$
$\max(0, -5 + 5x)$

Two classic (yet remarkable) results II

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

Breaking the curse of dimensionality: Barron (1993)

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

- More general theorems by Leshno *et al.* (1993) and Bach (2017).
- What about Chebyshev polynomials? Splines? Problems of convergence and extrapolation.
- There is another, yet more subtle curse of dimensionality.

Training the network

- θ is selected to minimize the quadratic error function $\mathcal{E}\left(\theta; \mathbf{Y}, \widehat{\mathbf{y}}\right)$:

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

- Where from do the observations $\mathbf{Y}$ come? Observed data vs. simulated epochs.
- How do we solve this minimization problem?
- Other objective functions are possible.

Back propagation

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

$$\begin{aligned}
\frac{\partial\mathcal{E}\left(\theta; y_j, \widehat{y}_j\right)}{\partial\theta_0} &= y_j - g\left(\mathbf{x}_j; \theta\right) \\
\frac{\partial\mathcal{E}\left(\theta; y_j, \widehat{y}_j\right)}{\partial\theta_m} &= \left(y_j - g\left(\mathbf{x}_j; \theta\right)\right)\phi\left(z_m\right), \text{ for } \forall m \\
\frac{\partial\mathcal{E}\left(\theta; y_j, \widehat{y}_j\right)}{\partial\theta_{0,m}} &= \left(y_j - g\left(\mathbf{x}_j; \theta\right)\right)\theta_m\phi'\left(z_m\right), \text{ for } \forall m \\
\frac{\partial\mathcal{E}\left(\theta; y_j, \widehat{y}_j\right)}{\partial\theta_{n,m}} &= \left(y_j - g\left(\mathbf{x}_j; \theta\right)\right)\theta_m x_n\phi'\left(z_m\right), \text{ for } \forall n, m
\end{aligned}$$

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

- The derivative $\phi'(z)$ will be trivial to evaluate if we use a ReLU.
- Back propagation will be particularly important below when we introduce multiple layers.

An approach to minimization

- One approach to optimization is to minimize a local model that approximates the true objective function.
- The local model can be a first- or second-order Taylor approximation of the objective function.
- For example, suppose a function $\mathcal{E}$ is roughly approximated as a quadratic form:

$$\mathcal{E}(\theta) \approx \frac{1}{2}\theta^T \mathbf{A}\theta - \mathbf{b}^T\theta + c$$

  where A is a square, symmetric, positive-definite matrix.
- Then $\mathcal{E}(\theta)$ is minimized by the solution to:

$$\mathbf{A}\theta = \mathbf{b}$$

- We can use this result to build a descent direction iteration if we know A and b (or we have approximations to them).

Descent direction iteration

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

Gradient descent method

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

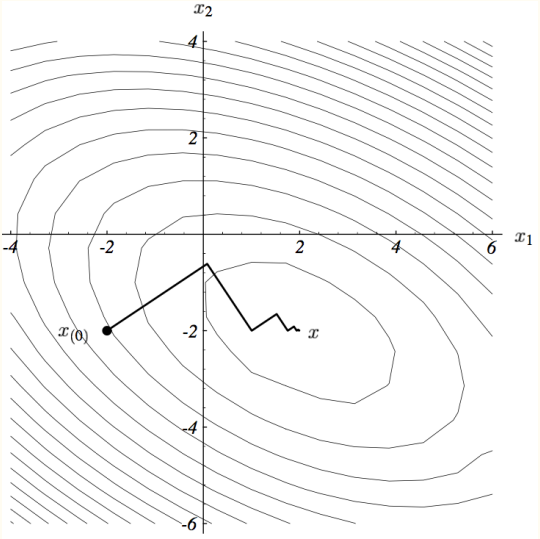
$$\mathbf{d}^{(k)} = -\frac{\nabla\mathcal{E}(\theta^{(k)})}{||\nabla\mathcal{E}(\theta^{(k)})||}$$

- One way to set the step size is to solve:

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

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

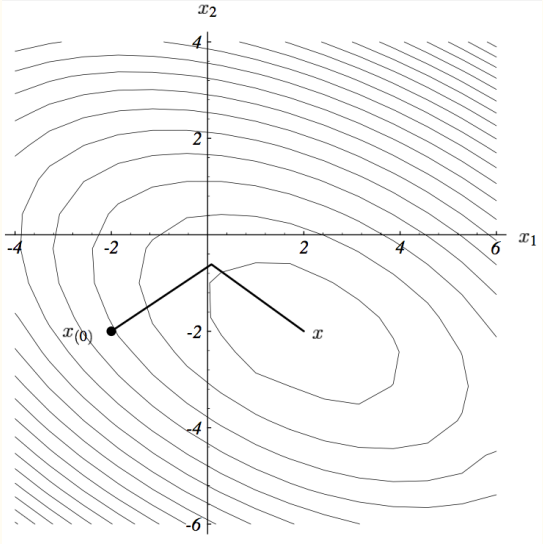
Steepest descent method



Conjugate descent method

- Gradient descent can perform poorly in narrow valleys (it may require many steps to make progress).
- Famous example: Rosenbrock function.
- The *conjugate gradient* method overcomes this problem by somehow constructing to be conjugate to the old gradient, and to all previous directions traversed.
- Define $g(\theta) = \nabla \mathcal{E}(\theta)$.
- In first iteration, set: $d^{(1)} = -g(\theta^{(1)})$ and $\theta^{(2)} = \theta^{(1)} + \alpha^{(1)}\mathbf{d}^{(1)}$. Here, $\alpha^{(1)}$ is arbitrary.
- Subsequent iterations set $\mathbf{d^{(k+1)}} = -g^{(k+1)} + \beta^{(k)}\mathbf{d^{(k)}}$.

Conjugate descent method



Conjugate descent method

- There are two approaches to set β:

  1. Fletcher-Reeves:
  $$\beta^{(k)} = \frac{g^{(k)T} g^{(k)}}{g^{(k-1)T} g^{(k-1)}}$$

  2. Olak-Ribiere:
  $$\beta^{(k)} = \frac{g^{(k)T} (g^{(k)} - g^{(k-1)})}{g^{(k-1)T} g^{(k-1)}}$$

- The Olak-Ribiere requires an automatic reset at every iteration: $\beta \leftarrow \max(\beta, 0)$.

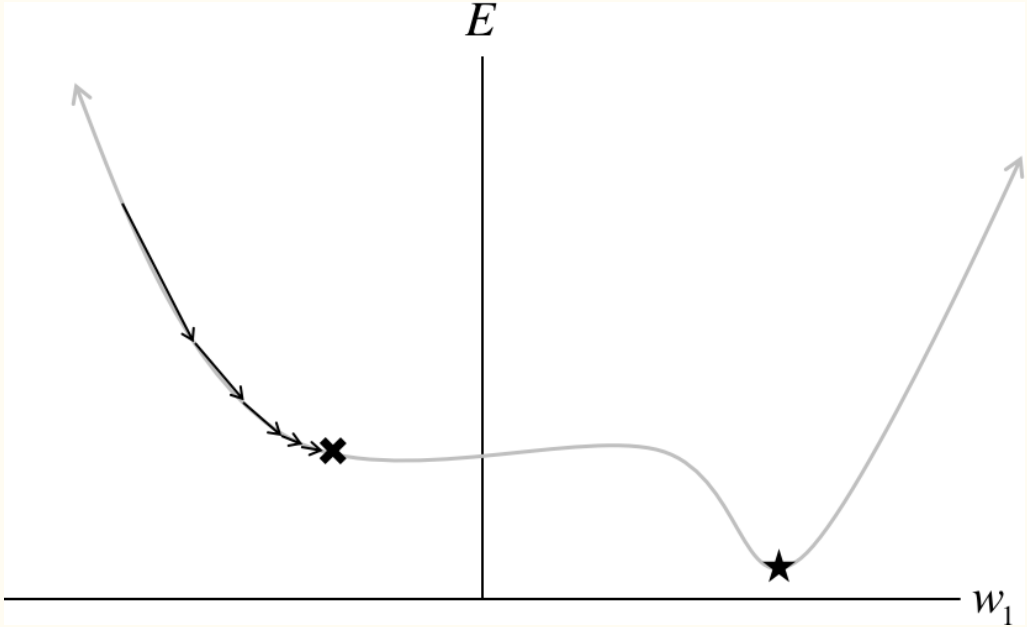
- If the function to minimize has flat areas, one can introduce a *momentum* update equation:
$$v^{(k+1)} = \beta v^{(k)} - \alpha g^{(k)}$$
$$\theta^{(k+1)} = \theta^{(k)} + v^{(k+1)}$$

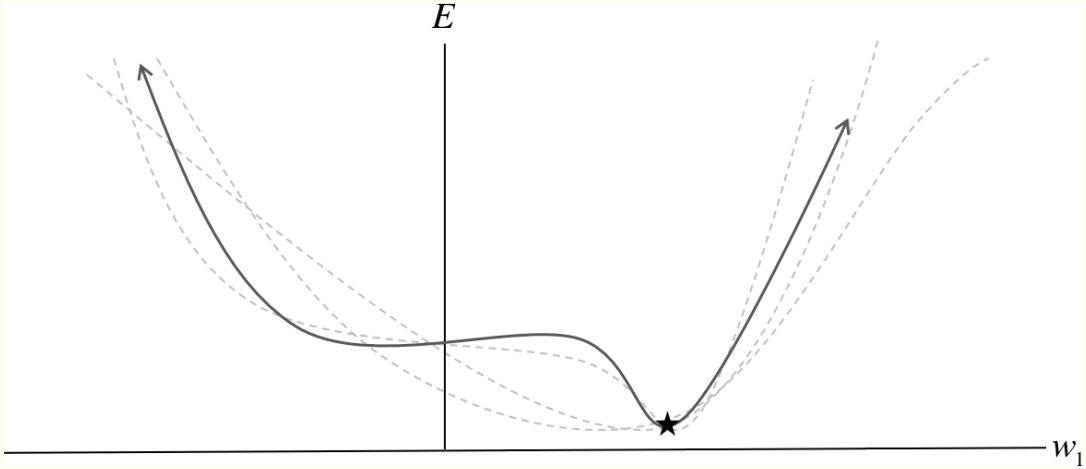
- The modification reverts to the gradient descent version if $\beta = 0$.

- Intuitively, the momentum update is like a ball rolling down an almost horizontal surface.

Stochastic gradient descent and minibatch

- Even with back propagation, evaluating the gradient for the whole training set can be costly.
- Stochastic gradient descent: Intuition from Monte Carlos.
- An additional advantage.
- A compromise between using the whole training set and pure stochastic gradient descent: minibatch gradient descent.
- This is the most popular algorithm to train neural networks.
- Intuition from GMM. Notice also resilience to scaling.
- In practice, we do not need a global min ($\neq$ likelihood).
- You can flush the algorithm to a graphics processing unit (GPU) or a tensor processing unit (TPU) instead of a standard CPU.


E
w_1

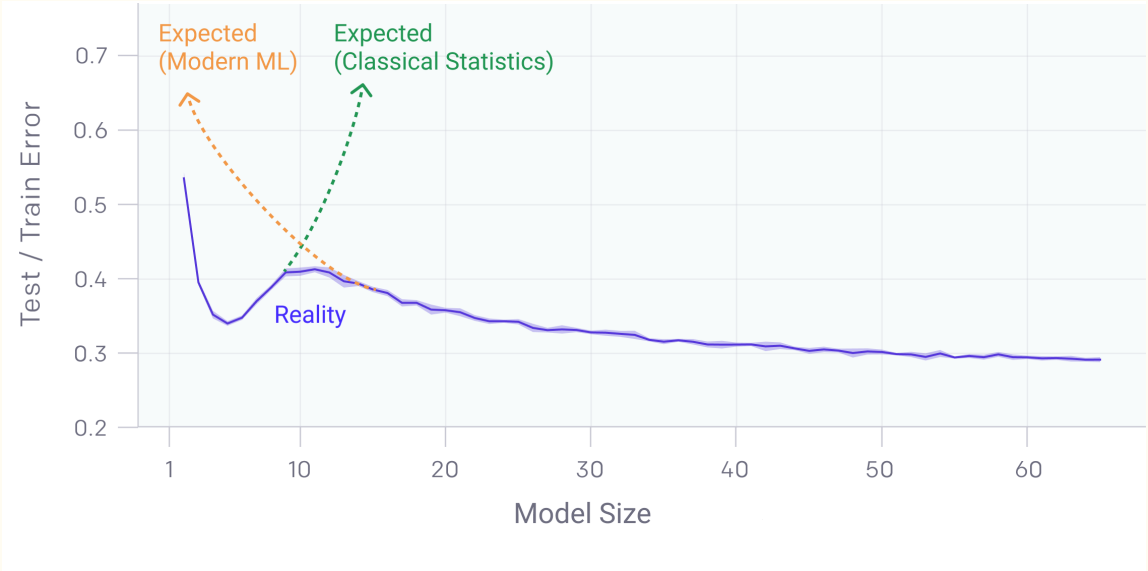

E
w_1

Alternative minimization algorithms

1. More sophisticated stochastic gradient descent: Adam (Adaptive Moment Estimation). It uses running averages of both the gradients and the second moments of the gradients.
2. Newton and Quasi-Newton methods are unlikely to be of much use in practice. Why?
3. McMc/Simulated annealing.
4. Genetic algorithms:
    - In fact, much of the research in deep learning incorporates some flavor of genetic selection.
    - Basic idea.

Further ideas

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


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

Multiple layers I

- The hidden layers can be multiplied without limit in a feed-forward ANN.
- We build K layers:

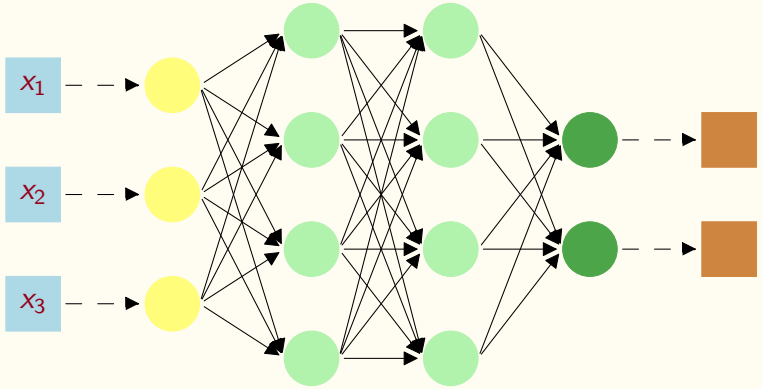
$$z_m^1 = \theta_{0,m}^1 + \sum_{n=1}^{N} \theta_{n,m}^1 x_n$$

and

$$z_m^2 = \theta_{0,m}^2 + \sum_{m=1}^{M} \theta_m^2 \phi\left(z_m^1\right)$$

...

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



Input Values Hidden Layer 1 Output Layer

Input Layer Hidden Layer 2

Multiple layers II

- Why do we want to introduce hidden layers?
  1. It works! Our brains have six layers. AlphaGo has 12 layers with ReLUs.
  2. Hidden layers induce highly nonlinear behavior.
  3. Allow for clustering of variables.
- We can have different M's in each layer $\Rightarrow$ fewer neurons in higher layers allow for compression of learning into fewer features.
- We can also add multidimensional outputs.
- Or even to produce, as output, a probability distribution, for example, using a softmax layer:

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

Application to Economics

- Our goal is to solve the recursive continuous-time Hamilton-Jacobi-Bellman (HJB) equation globally:

$$\rho V(\mathbf{x}) = \max_{\boldsymbol{\alpha}} r(\mathbf{x}, \boldsymbol{\alpha}) + \nabla_x V(\mathbf{x}) f(\mathbf{x}, \boldsymbol{\alpha}) + \frac{1}{2} tr(\sigma(\mathbf{x}))^T \Delta_x V(\mathbf{x}) \sigma(\mathbf{x}))$$

$$\text{s.t.} \quad G(\mathbf{x}, \boldsymbol{\alpha}) \leq \mathbf{0} \quad \text{and} \quad H(\mathbf{x}, \boldsymbol{\alpha}) = \mathbf{0}$$

- Think about the cases where we have many state variables.
- Alternatives for this solution?

Neural networks

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

Error criterion I

- The HJB error:

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

- The policy function error:

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

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

Error criterion II

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

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

  the dual feasibility error:

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

  and the complementary slackness error:

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

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

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

Training

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

An Example

The continuous-time neoclassical growth model I

- We start with the continuous-time neoclassical growth model because it has closed-form solutions for the policy functions, which allows us to focus our attention on the analysis of the value function approximation.
- We can then back out the policy function from this approach and compare it to the results of the next step in which we approximate the policy functions themselves with a neural net.
- A single agent deciding to either save in capital or consume with a HJB equation :

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

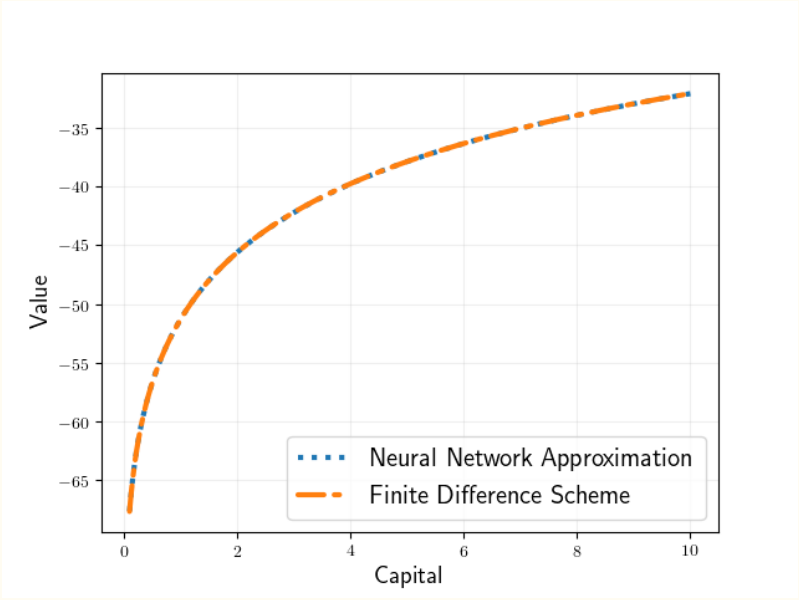
- Notice that $c = (U')^{-1}(V'(k))$. With CRRA utility, this simplifies further to $c = (V'(k))^{-\frac{1}{\gamma}}$.
- We set $\gamma = 2$, $\rho = 0.04$, $F(k) = 0.5 * k^{0.36}$, $\delta = 0.05$.

The continuous-time neoclassical growth model II

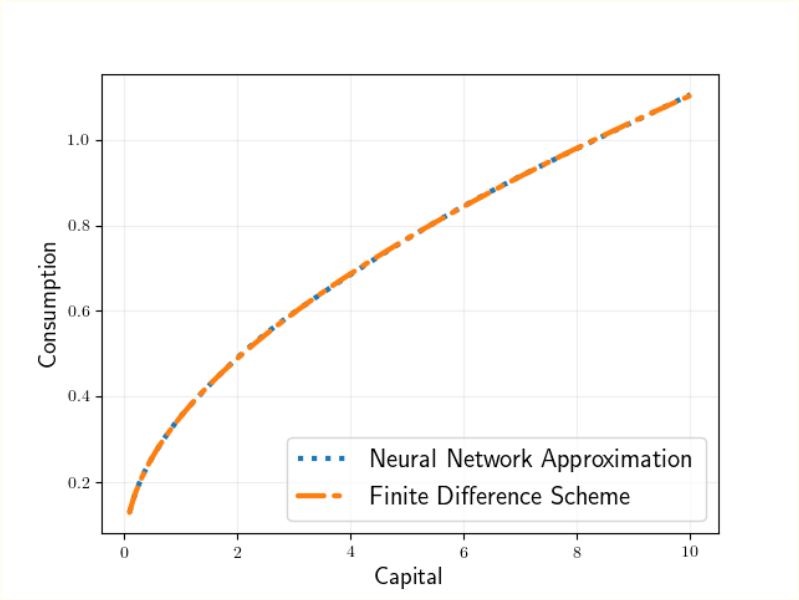
- We approximate the value function $V(k)$ with a neural network, $\tilde{V}(k;\Theta)$ with an "HJB error":

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

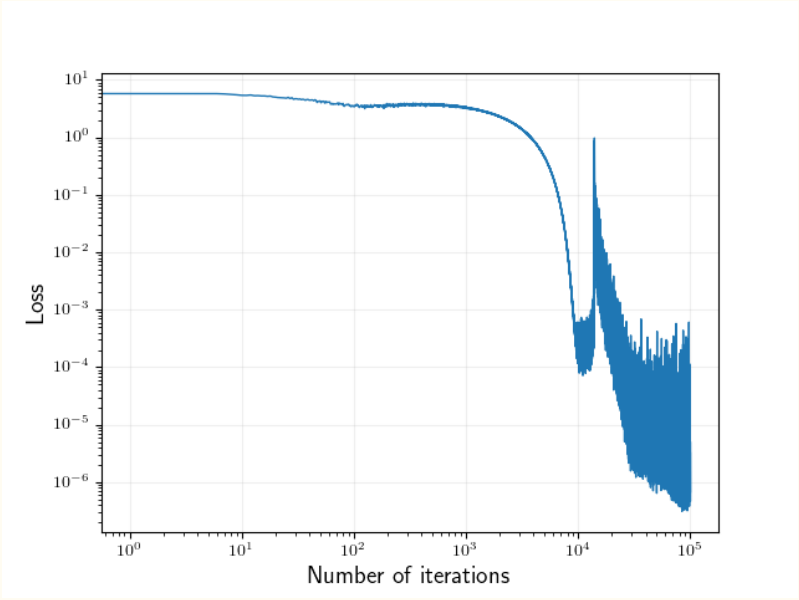
- Details:
  1. 3 layers.
  2. 8 neurons per layers.
  3. $\tanh(x)$ activation.
  4. Normal initialization $\mathcal{N}\left(0, 4\sqrt{\frac{2}{n_{input}+n_{output}}}\right)$ with input normalization.



(a) Value with closed-form policy



(c) Consumption with closed-form policy



(e) HJB error with closed-form policy

Approximating the policy function

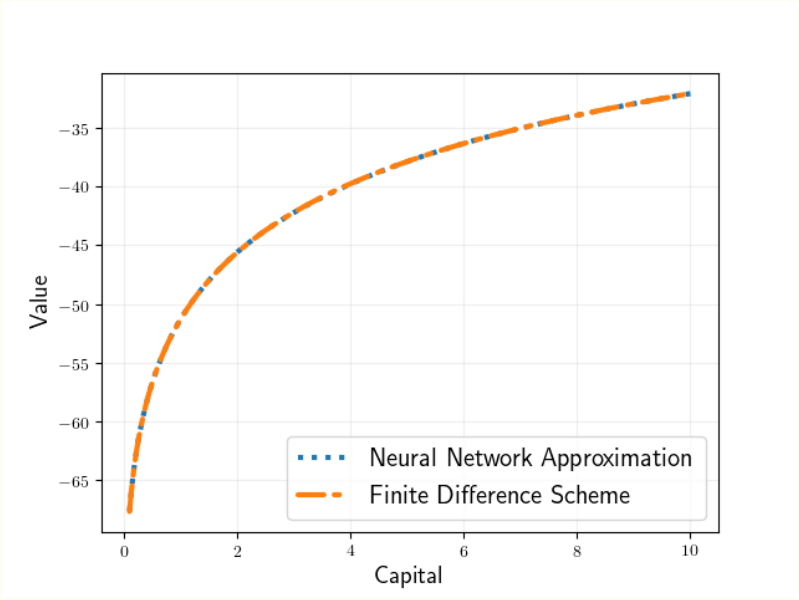
- Let us not use the closed-form consumption policy function but rather approximate said policy function directly with a policy neural network $\tilde{C}(k;\Theta^C)$.

- The new HJB error:

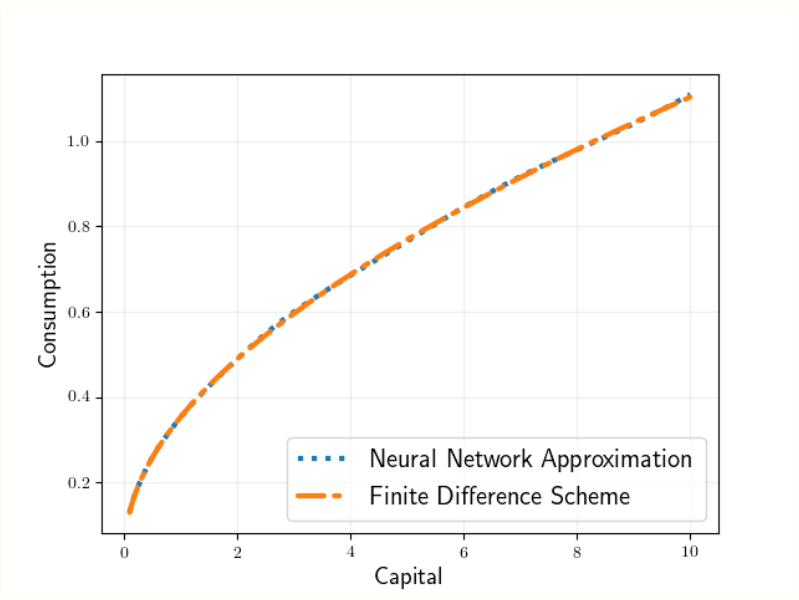
$$err_{HJB} = \rho\tilde{V}(k;\Theta^V) - U\left(\tilde{C}(k;\Theta^C)\right) - \frac{\partial\tilde{V}(k;\Theta^V)}{\partial k}\left[F(k) - \delta * k - \tilde{C}(k;\Theta^C)\right]$$

- Now we have a policy function error:

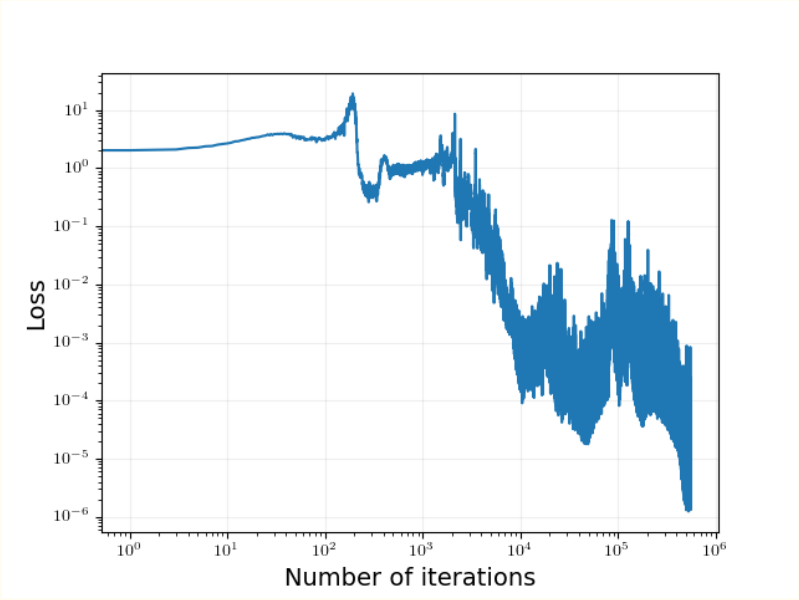
$$err_C = (U')^{-1}\left(\frac{\partial\tilde{V}(k;\Theta^V)}{\partial k}\right) - \tilde{C}(k;\Theta^C)$$



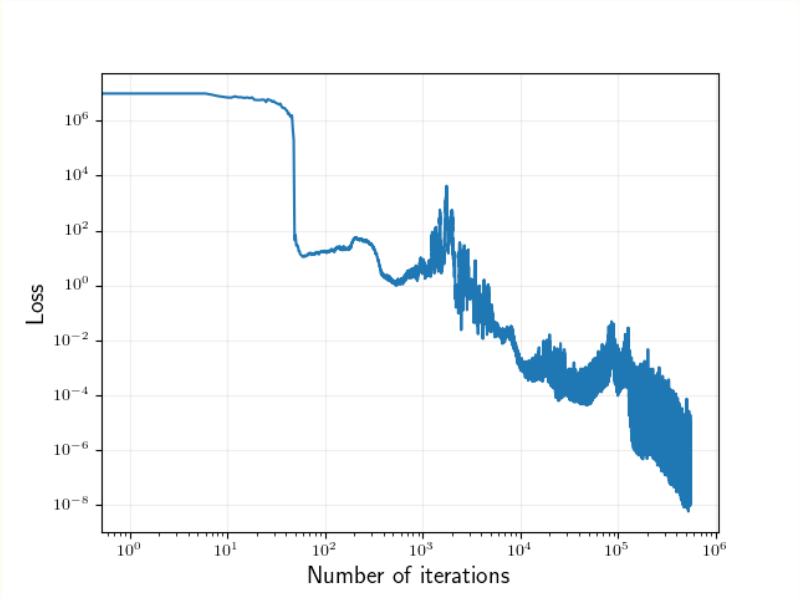
(b) Value with policy approximation



(d) Consumption with policy approximation



(f) HJB error with policy approximation

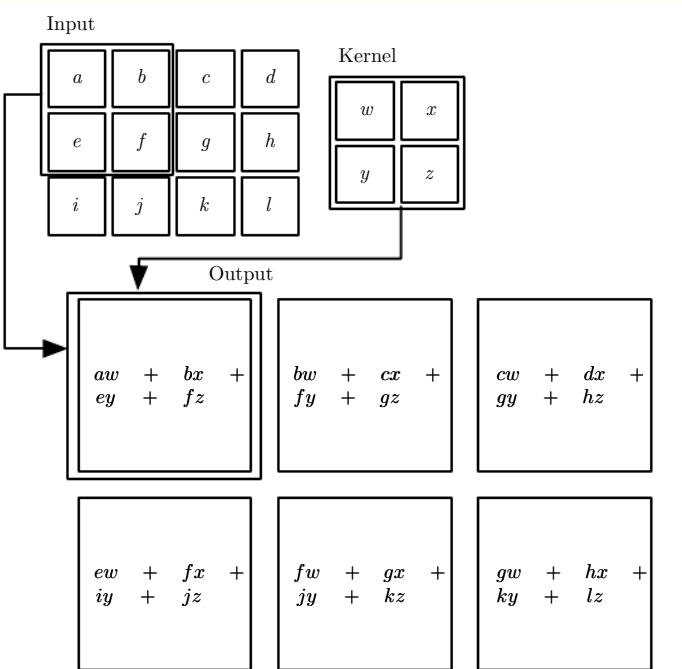


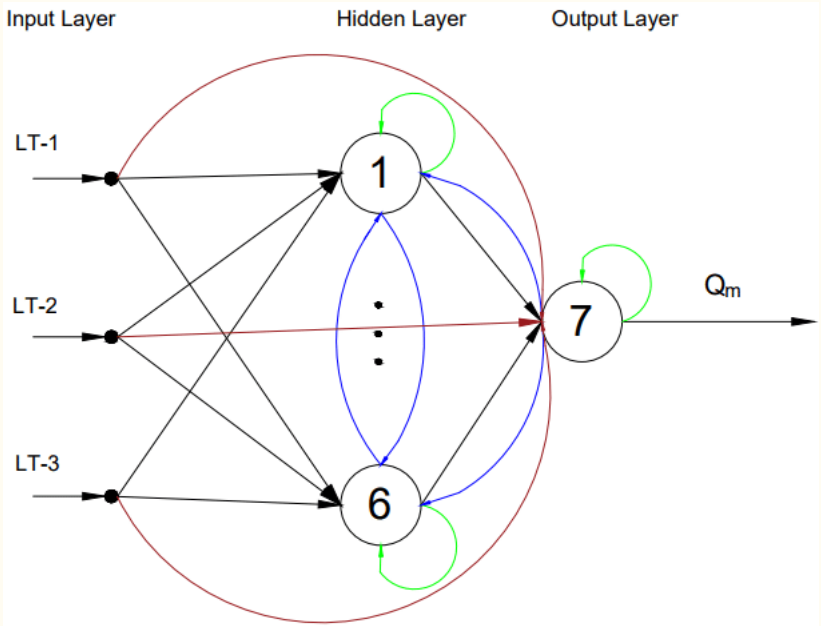
(g) Policy error with policy approximation

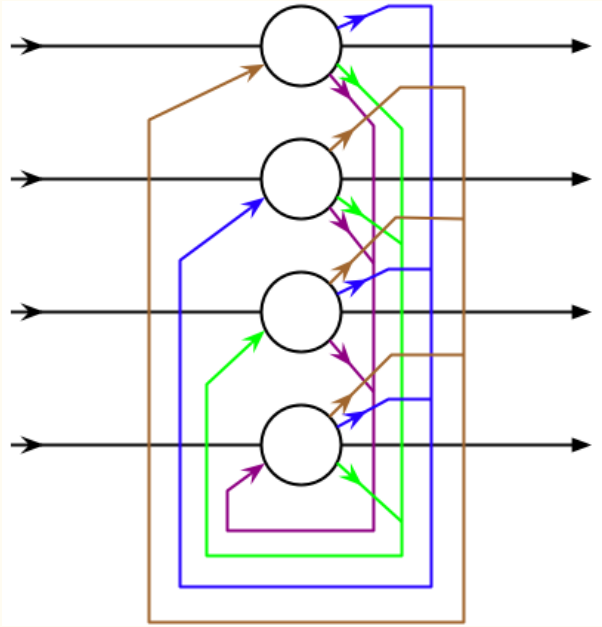
Alternative ANNs

Alternative ANNs

- Convolutional neural networks.
- Feedback ANN such as the Hopfield network.
- Self-organizing maps (SOM).
- ANN and reinforcement learning.


Input
a
b
c
d
e
f
g
h
i
j
k
l
Kernel
w
x
y
z
Output
aw + bx + ey + fz
bw + cx + fy + gz
cw + dx + gy + hz
ew + fx + iy + jz
fw + gx + jy + kz
gw + hx + ky + lz


Input Layer
Hidden Layer
Output Layer
LT-1
LT-2
LT-3
1
6
7
Q_m



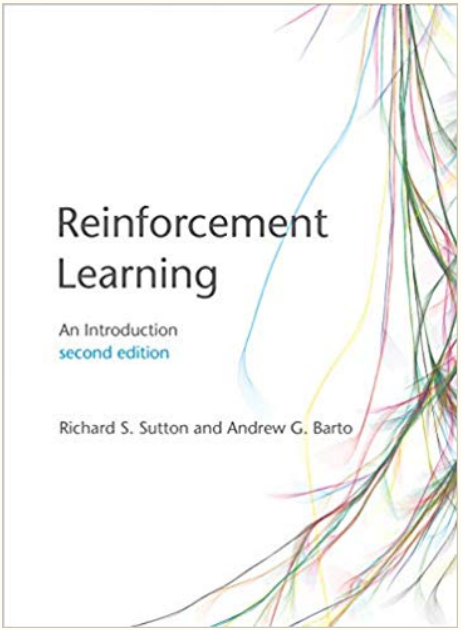
Reinforcement learning

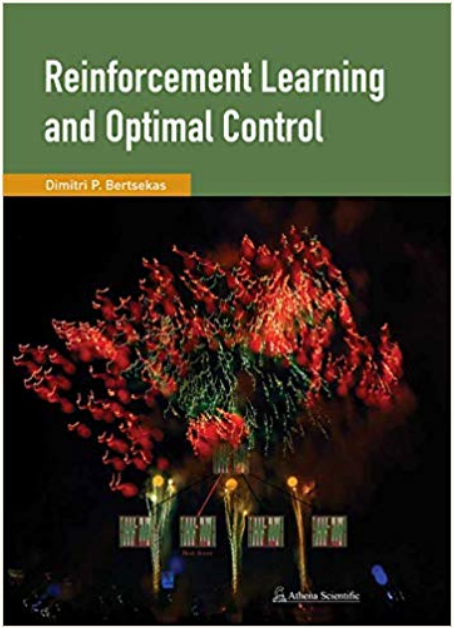
Reinforcement learning

- Main idea: Algorithms that use training information that evaluates the actions taken instead of deciding whether the action was correct.
- Purely *evaluative feedback* to assess how good the action taken was, but not whether it was the best feasible action.
- Useful when:
  1. The dynamics of the state is unkown but simulation is easy: model-free vs. model-based reinforcement learning.
  2. Or the dimensionality is so high that we cannot store the information about the DP in a table.
- Work surprisingly well in a wide range of situations, although no methods that are guaranteed to work.
- Key for success in economic applications: ability to simulate fast (link with massive parallelization). Also, it complements very well with neural networks.

Comparison with alternative methods

- Similar (same?) ideas are called approximate dynamic programming or neuro-dynamic programming.
- Traditional dynamic programming: we optimize over best feasible actions.
- Supervised learning: purely *instructive feedback* that indicates best feasible action regardless of action actually taken.
- Unsupervised learning: hard to use for optimal control problems.
- In practice, we mix different methods.
- Current research challenge: how do we handle associate behavior effectively?


Reinforcement
Learning
An Introduction
second edition
Richard S. Sutton and Andrew G. Barto


Reinforcement Learning
and Optimal Control
Dimitri P. Bertsekas
Athena Scientific

Example: Multi-armed bandit problem

- You need to choose action a among k available options.
- Each option is associated with a probability distribution of payoffs.
- You want to maximize the expected (discounted) payoffs.
- But you do not know which action is best, you only have estimates of your value function (dual control problem of identification and optimization).
- You can observe actions and period payoffs.
- Go back to the study of "sequential design of experiments" by Thompson (1933, 1934) and Bellman (1956).


LIBERTY BELL
Three Bells
Straight Flush of
Flush of
16
Flush of
12
Two
Two
8
4
2
SLOT-MACHI

Theory vs. practice

- You can follow two pure strategies:
  1. Follow *greedy* actions: actions with highest expected value. This is known as *exploiting*.
  2. Follow *non-greedy* actions: actions with dominated expected value. This is known as *exploring*.
- This should remind you of a basic dynamic programming problem: what is the optimal mix of pure strategies?
- If we impose enough structure on the problem (i.e., distributions of payoffs belong to some family, stationarity, etc.), we can solve (either theoretically or applying standard solution techniques) the optimal strategy (at least, up to some upper bound on computational capabilities).
- But these structures are too restrictive for practical purposes outside the pages of *Econometrica*.

A policy-based method I

- Proposed by Thathachar and Sastry (1985).
- A very simple method that uses the averages $Q_n(a)$ of rewards $R_i(a), i = \{1, ..., n\}$, actually received:

$$Q_n(a) = \frac{1}{n} \sum_{i=1}^{n-1} R_i(a)$$

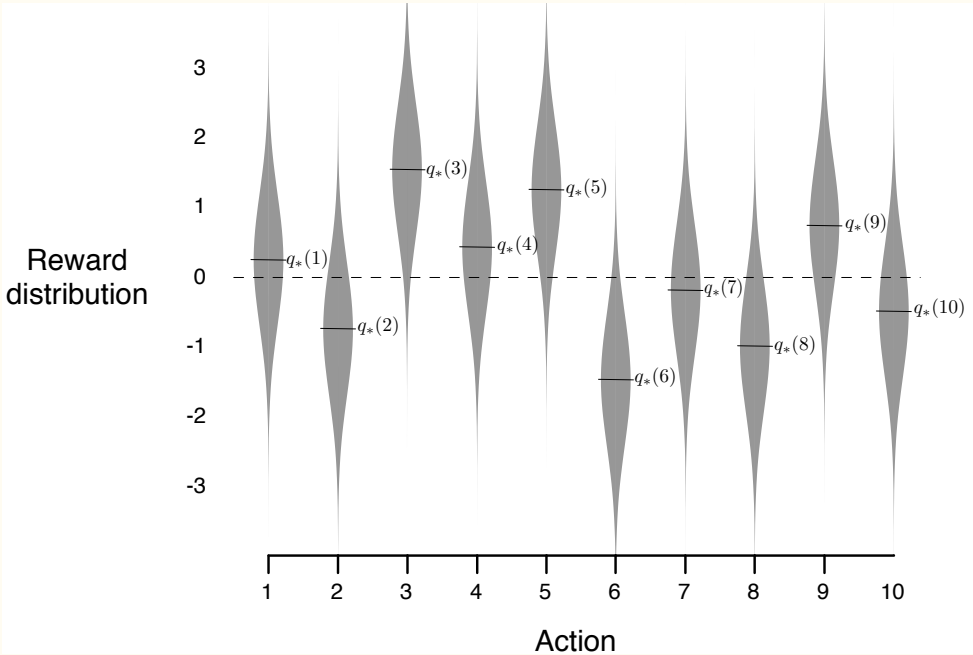
- We start with $Q_0(a) = 0$ for all k. Here (and later), we randomize among ties.
- We update $Q_n(a)$ thanks to the nice recursive update based on linearity of means:

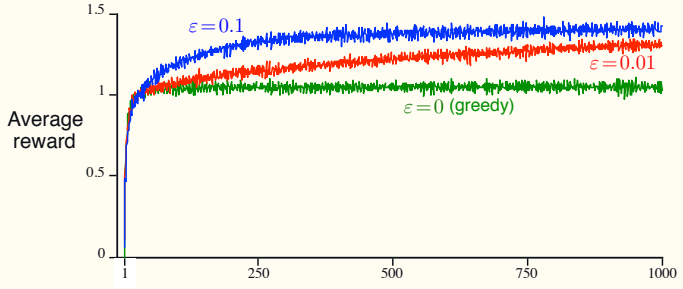
$$Q_{n+1}(a) = Q_n(a) + \frac{1}{n} [R_n(a) - Q_n(a)]$$

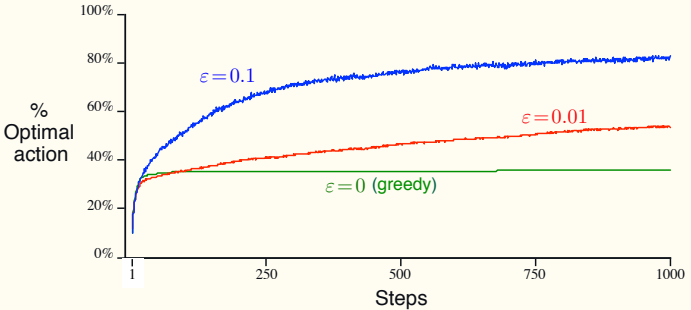
Averages of actions not picked are not updated.

A policy-based method II

- How do we pick actions?
  1. Pure greedy method: $\arg\max_a Q_t(a)$.
  2. ϵ-greedy method. Mixed best action with a random trembling.
- Easy to generalize to more sophisticated strategies.
- In particular, we can connect with genetic algorithms (AlphaGo).


Reward distribution
3
2
1
0
-1
-2
-3
$q_*(1)$
$q_*(2)$
$q_*(3)$
$q_*(4)$
$q_*(5)$
$q_*(6)$
$q_*(7)$
$q_*(8)$
$q_*(9)$
$q_*(10)$
1
2
3
4
5
6
7
8
9
10
Action


1.5
1
0.5
0
Average reward
$\varepsilon = 0.1$
$\varepsilon = 0.01$
$\varepsilon = 0$ (greedy)
1
250
500
750
1000
Steps


100%
80%
60%
40%
20%
0%
% Optimal action
$\varepsilon = 0.1$
$\varepsilon = 0.01$
$\varepsilon = 0$ (greedy)
1
250
500
750
1000
Steps

A more general update rule

- Let's think about a modified update rule:

$$Q_{n+1}(a) = Q_n(a) + \alpha \left[R_n(a) - Q_n(a) \right]$$

  for $\alpha \in (0, 1]$.
- This is equivalent, by recursive substitution, to:

$$Q_{n+1}(a) = (1-\alpha)^n Q_1(a) + \alpha \sum_{i=1}^{n-1} \alpha (1-\alpha)^{n-i} R_i(a)$$

- We can also have a time-varying $\alpha_n(a)$, but, to ensure convergence with probability 1 as long as:

$$\sum_{i=1}^{\infty} \alpha_n(a) = \infty$$

$$\sum_{i=1}^{\infty} \alpha_n^2(a) = \infty$$

Improving the algorithm

- We can start with "optimistic" Q_0 to induce exploration.
- We can implement an upper-confidence-bound action selection

$$\arg\max_a \left[Q_n(a) + c\sqrt{\frac{\log n}{N_n(a)}} \right]$$

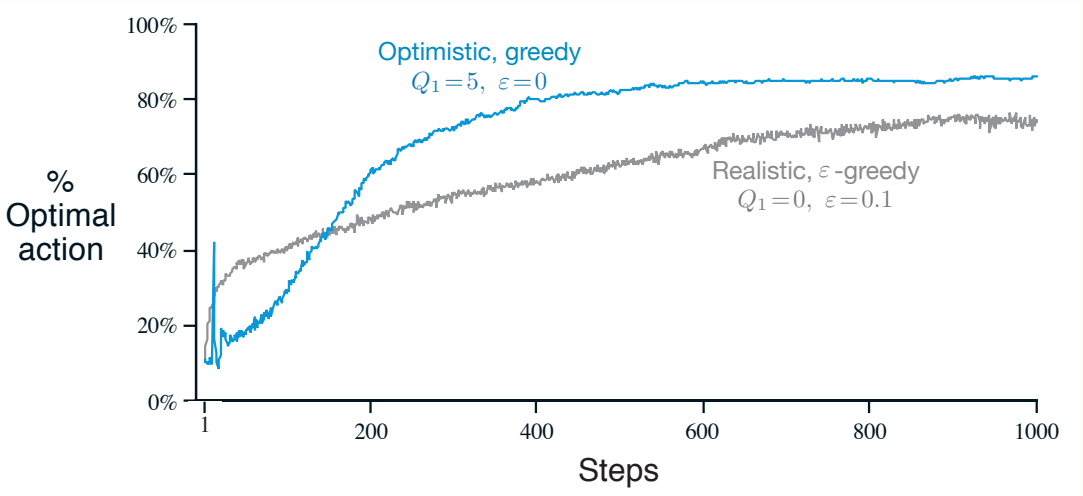
- We can have a gradient bandit algorithms based on a softmax choice:

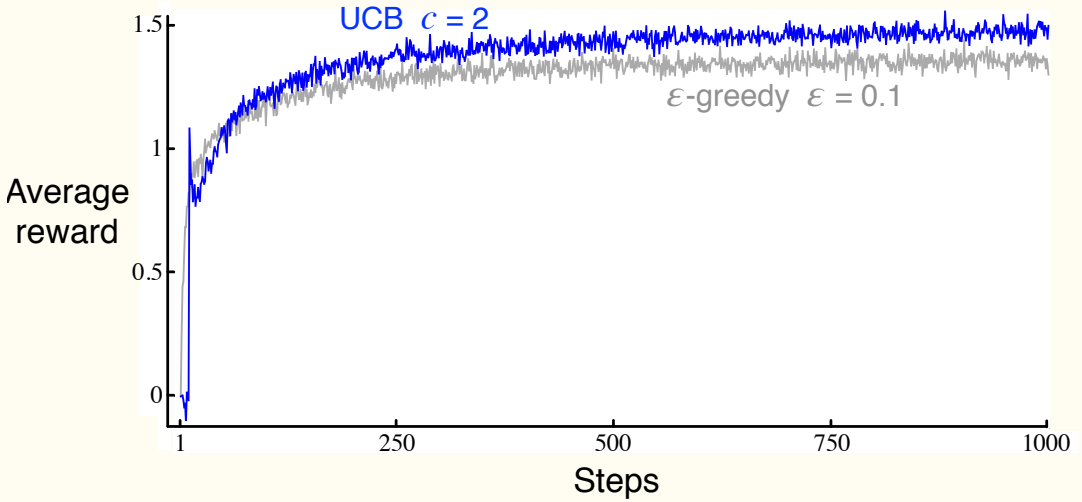
$$\pi_n(a) = \mathrm{P}(A_n = a) = \frac{e^{H_n(a)}}{\sum_{b=1}^{k} e^{H_n(b)}}$$

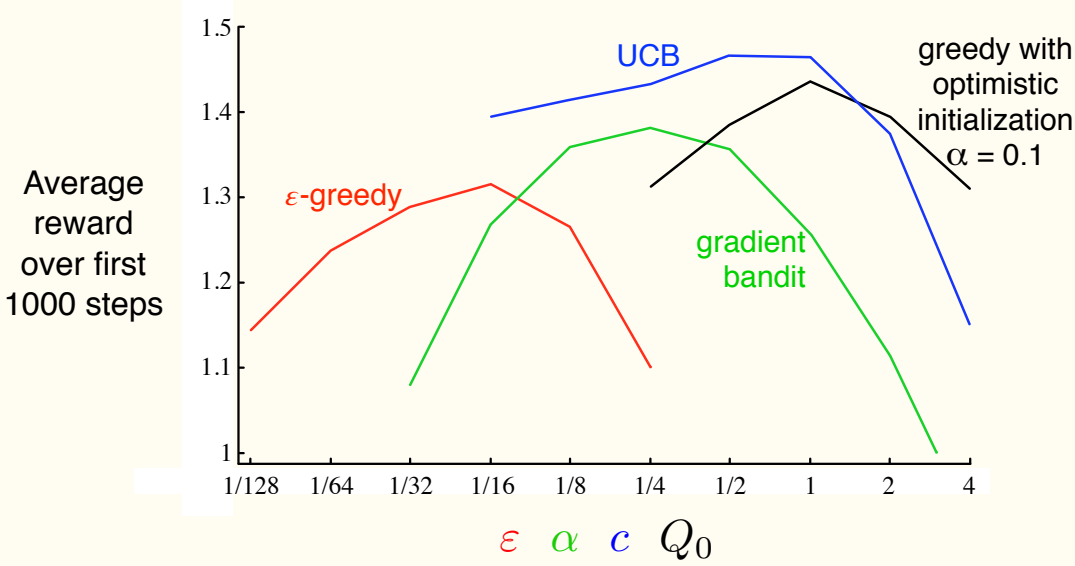
where

$$\begin{aligned} H_{n+1}(A_n) &= H_n(A_n) + \alpha(1 - \pi_n(A_n))\left(R_n(a) - \overline{R}_n\right) \\ H_{n+1}(a) &= H_n(a) - \alpha\pi_n(a)\left(R_n(a) - \overline{R}_n\right) \text{ for all } a \neq A_n \end{aligned}$$

This is a slightly hidden version of a stochastic gradient algorithm that we will see soon when we talk about deep learning.


% Optimal action
100%
80%
60%
40%
20%
0%
Optimistic, greedy
$Q_1 = 5,\ \varepsilon = 0$
Realistic, ε-greedy
$Q_1 = 0,\ \varepsilon = 0.1$
1
200
400
600
800
1000
Steps


UCB $c = 2$
ε-greedy $\varepsilon = 0.1$
Average reward
1.5
1
0.5
0
1
250
500
750
1000
Steps


Average reward over first 1000 steps
1.5
1.4
1.3
1.2
1.1
1
UCB
greedy with optimistic initialization
α = 0.1
ε-greedy
gradient bandit
1/128
1/64
1/32
1/16
1/8
1/4
1/2
1
2
4
ε α c Q_0

Other algorithms

- Monte Carlo prediction.
- Temporal-difference (TD) learning:

$$V^{n+1}(s_t) = V^n(s_t) + \alpha\left(r_{t+1} + \beta V^n(s_{t+1}) - V^n(s_t)\right)$$

- SARSA $\Rightarrow$ On-policy TD control:

$$Q^{n+1}(a_t, s_t) = Q^n(a_t, s_t) + \alpha\left(r_{t+1} + \beta Q^n(a_{t+1}, s_{t+1}) - Q^n(a_t, s_t)\right)$$

- Q-learning $\Rightarrow$ Off-Policy TD Control:

$$Q^{n+1}(a_t, s_t) = Q^n(a_t, s_t) + \alpha\left(r_{t+1} + \beta \max_{a_{t+1}} Q^n(a_{t+1}, s_{t+1}) - Q^n(a_t, s_t)\right)$$

- Value-based methods.
- Actor-critic methods.