Deep learning and reinforcement learning

Jesús Fernández-Villaverde\textsuperscript{1} and Galo Nuño\textsuperscript{2}

October 15, 2021

\textsuperscript{1}University of Pennsylvania

\textsuperscript{2}Banco de España
A short introduction
The problem

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

\[ y = f(x) \]

where \( y \) is a scalar and \( x = \{x_1, x_2, ..., x_N\} \) 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, \( f(x) \) 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 $f(x)$ built as a linear combination of $M$ generalized linear models of $x$ of the form:

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

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 $\theta$ such that $g^{NN}(x; \theta)$ is as close to $f(x)$ as possible given some relevant metric (e.g., $L^2$ norm).

- This is known as “training” the network.
Comparison with other approximations

- Compare:

\[ y \simeq g^{NN}(x; \theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m \left( \theta_{0,m} + \sum_{n=1}^{N} \theta_{n,m} x_n \right) \]

with a standard projection:

\[ y \simeq g^{CP}(x; \theta) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x) \]

where \( \phi_m \) 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 \textit{multilayer} composition of $J > 1$ neural networks:

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

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 $\theta$ such that $g^{DL}(x; \theta)$ approximates a target function $f(x)$ 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 multiprocessing.

- Thus, neural networks have considerable option value as solution methods in economics.
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.
• Big splash: AlphaGo vs. Lee Sedol in March 2016.

• *Silver et al.* (2018): now applied to chess, shogi, Go, and StarCraft II.

• Check also:


• Very different than Deep Blue against Kasparov.

• New and surprising strategies.

• However, you need to keep this accomplishment in perspective.
A reinforcement learning (RL) policy network trained to predict human expert moves in a data set of positions.

- **Rollout policy** $\pi(s)$
- **SL policy network** $\sigma(s)$
- **RL policy network** $\rho(s)$
- **Value network** $v(\theta)$

**Policy gradient**: $\nabla_{\theta} v(\theta)$ where $v(\theta)$ is the value function that maximizes the expected outcome $\mathbb{E}[\pi|s]$. The policy gradient is used to update the policy network $\rho(s)$.

**Training pipeline**:
1. **First stage**: Human expert positions and self-play positions are used to train the policy network $\rho(s)$ and the value network $v(\theta)$.
2. **Second stage**: The policy network is improved by policy gradient reinforcement learning (RL).
3. **Final stage**: A value network is trained to estimate the value of positions from the self-play data set.

**Evaluation**:
- **Mean squared error** on expert games
- **AlphaGo win rate** (%)

**Extended Data Table 3**: Small improvements in accuracy led to large changes in playing strength and accuracy of policy and value networks.
In this paper, we generalize this approach into a single AlphaZero algorithm that can achieve superhuman performance in the game of Go by reinforcement learning from self-play. We apply AlphaZero to the games of chess and shogi, as well as Go, by using the same algorithm trained solely by reinforcement learning from games of self-play; these are then used to guide its search through a tree for each state. Each search consists of a series of simulated moves, and AlphaZero learns the outcome of each move through simulation. AlphaZero then uses a tree search algorithm, a general-purpose reinforcement learning algorithm, and a general-purpose neural network to compute the expected outcome of each state.

In chess, the strongest programs are based on a combination of sophisticated search techniques, including heuristics and network architecture for all three games. These programs evaluate positions by an algorithm similar to those used by computer chess programs, again based on a highly general-purpose architecture. In shogi, the strongest programs, such as the 2017 Computer Shogi Association (CSA) world champion Stockfish, use a large number of clever heuristics and domain-specific adaptations. These augmentations, focusing on the 2016 Top Chess Engine Championship (TCEC) season 9, are substantial human effort, whereas general intelligence researchers, culminating in high-performance grand challenge task for a generation of artificial intelligence researchers.

In Go, the game with the longest history in the history of artificial intelligence, the strongest programs are based on a combination of sophisticated search techniques, including heuristics and network architecture for all three games. These programs evaluate positions by an algorithm similar to those used by computer chess programs, again based on a highly general-purpose architecture. In shogi, the strongest programs, such as the 2017 Computer Shogi Association (CSA) world champion Stockfish, use a large number of clever heuristics and domain-specific adaptations. These augmentations, focusing on the 2016 Top Chess Engine Championship (TCEC) season 9, are substantial human effort, whereas general intelligence researchers, culminating in high-performance grand challenge task for a generation of artificial intelligence researchers.

Fig. 1. Training AlphaZero for 700,000 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.
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.
• \( N \) observables: \( x_1, x_2, \ldots, x_N \). We stack them in \( x \).

• Coefficients (or weights): \( \theta_0 \) (a constant), \( \theta_1, \theta_2, \ldots, \theta_N \). We stack them in \( \theta \).

• 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:

\[
    y = g(x; \theta) = \phi(z)
\]

The activation function might have some coefficients \( \gamma \) on its own.

• Why do we need an activation function?
Flow representation

Inputs

\[ x_1, x_2, x_3, \ldots, x_n \]

Weights

\[ \theta_1, \theta_2, \theta_3, \ldots, \theta_n \]

Net input

\[ \sum_{i=1}^{n} \theta_i x_i \]

Activation

\[ \gamma \]

Perceptron classification output
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: $\frac{1}{1+e^{-x}}$ activation function
Hyperbolic tangent: \( \frac{e^{2x} - 1}{e^{2x} + 1} \) activation function
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)
**Interpretation**

- $\theta_0$ controls the activation threshold.

- The level of the $\theta_i$'s for $i > 0$ control the activation rate (the higher the $\theta_i$'s, the harder the activation).

- Some textbooks separate the activation threshold and scaling coefficients from $\theta$ as different coefficients in $\phi$, but such separation moves notation farther away from standard econometrics.

- Potential identification problem between $\theta$ and more general activation functions with their own parameters.

- But in practice $\theta$ 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: $\max(0, \theta_{i,0} + \theta_{i,1}x)$

- $\max(0, x)$
- $\max(0, 1 + x)$
- $\max(0, -1 + x)$
- $\max(0, 0.5x)$
- $\max(0, 1.5x)$
- $\max(0, -x)$
- $-\max(0, x)$
- $-\max(0, -x)$
- $-1.5\max(0, -x)$
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:

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

- 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

**Borel measurable function**

A map $f : X \rightarrow Y$ 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.
$x^3 + x^2 - x - 1$
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 $O(1/M)$, where $M$ is the number of nodes. In comparison, for series approximations, the integrated square error is of order $O(1/(M^2/N))$ where $N$ is the dimensions of the function to be approximated.

- More general theorems by Leshno et al. (1993) and Bach (2017).


- There is another, yet more subtle curse of dimensionality.
• $\theta$ is selected to minimize the quadratic error function $E(\theta; Y, \hat{y})$:

$$
\theta^* = \arg \min_{\theta} E(\theta; Y, \hat{y}) = \arg \min_{\theta} \sum_{j=1}^{J} E(\theta; y_j, \hat{y}_j) = \arg \min_{\theta} \frac{1}{2} \sum_{j=1}^{J} \|y_j - g(x_j; \theta)\|^2
$$

• Where from do the observations $Y$ come? Observed data vs. simulated epochs.

• How do we solve this minimization problem?

• Other objective functions are possible.
In general, we can easily calculate $E(\theta^*; Y, \hat{y})$ and $\nabla E(\theta^*; Y, \hat{y})$ for a given $\theta^*$.

In particular, for the gradient, we can use back propagation (Rumelhart et al., 1986):

\[
\frac{\partial E(\theta; y_j, \hat{y}_j)}{\partial \theta_0} = y_j - g(x_j; \theta)
\]

\[
\frac{\partial E(\theta; y_j, \hat{y}_j)}{\partial \theta_m} = (y_j - g(x_j; \theta)) \phi(z_m), \text{ for } \forall m
\]

\[
\frac{\partial E(\theta; y_j, \hat{y}_j)}{\partial \theta_{0,m}} = (y_j - g(x_j; \theta)) \theta_m \phi'(z_m), \text{ for } \forall m
\]

\[
\frac{\partial E(\theta; y_j, \hat{y}_j)}{\partial \theta_{n,m}} = (y_j - g(x_j; \theta)) \theta_m x_n \phi'(z_m), \text{ for } \forall n, m
\]

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 $E$ is roughly approximated as a quadratic form:

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

where $A$ is a square, symmetric, positive-definite matrix.

- Then $E(\theta)$ is minimized by the solution to:

$$A \theta = b$$

- 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 $d^{(k)}$ using local information such as gradient or Hessian.
  3. Compute step size $\alpha^{(k)}$.
  4. Compute the next candidate point: $\theta^{(k+1)} \leftarrow \theta^{(k)} + \alpha^{(k)} d^{(k)}$.

- Choice of $\alpha$ and $d$ 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 \( g(\theta) = \nabla E(\theta) \).
- In first iteration, set: \( d^{(1)} = -g(\theta^{(1)}) \) and \( \theta^{(2)} = \theta^{(1)} + \alpha^{(1)}d^{(1)} \). Here, \( \alpha^{(1)} \) is arbitrary.
- Subsequent iterations set \( d^{(k+1)} = -g^{(k+1)} + \beta^{(k)}d^{(k)} \).
Conjugate descent method
Conjugate descent method

• There are two approaches to set $\beta$:

  1. Fletcher-Reeves:

$$\beta^{(k)} = \frac{g^{(k)} g^{(k)}}{g^{(k-1)T}g^{(k-1)}}$$

  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.
Figure 2-6. Batch gradient descent is sensitive to saddle points, which can lead to premature convergence.

We only have a single weight, and we use random initialization and batch gradient descent to find its optimal setting. The error surface, however, has a flat region (also known as saddle point in high-dimensional spaces), and if we get unlucky, we might find ourselves getting stuck while performing gradient descent.

Another potential approach is stochastic gradient descent (SGD), where at each iteration, our error surface is estimated only with respect to a single example. This approach is illustrated by Figure 2-7, where instead of a single static error surface, our error surface is dynamic. As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions.
Batch gradient descent is sensitive to saddle points, which can lead to premature convergence. We only have a single weight, and we use random initialization and batch gradient descent to find its optimal setting. The error surface, however, has a flat region (also known as saddle point in high-dimensional spaces), and if we get unlucky, we might find ourselves getting stuck while performing gradient descent.

Another potential approach is stochastic gradient descent (SGD), where at each iteration, our error surface is estimated only with respect to a single example. This approach is illustrated by Figure 2-7, where instead of a single static error surface, our error surface is dynamic. As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions.
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?


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.
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

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

... 

$$y \approx g(x; \theta) = \theta_0^K + \sum_{m=1}^{M} \theta^K_m \phi (z_{m-1}^K)$$
• 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:

$$y_m = \frac{e^{z_{m-1}}}{\sum_{m=1}^{M} e^{z_{m-1}}}$$
Application to Economics
Our goal is to solve the recursive continuous-time Hamilton-Jacobi-Bellman (HJB) equation globally:

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

s.t.  

$$G(x, \alpha) \leq 0$$  

and  

$$H(x, \alpha) = 0$$

Think about the cases where we have many state variables.

Alternatives for this solution?
Neural networks

- We define four neural networks:
  
  1. $\tilde{V}(x; \Theta^V): \mathbb{R}^N \rightarrow \mathbb{R}$ to approximate the value function $V(x)$.
  
  2. $\tilde{\alpha}(x; \Theta^\alpha): \mathbb{R}^N \rightarrow \mathbb{R}^M$ to approximate the policy function $\alpha$.
  
  3. $\tilde{\mu}(x; \Theta^\mu): \mathbb{R}^N \rightarrow \mathbb{R}^{L_1}$, and $\tilde{\lambda}(x; \Theta^\lambda): \mathbb{R}^N \rightarrow \mathbb{R}^{L_2}$ to approximate the Karush-Kuhn-Tucker (KKT) multipliers $\mu$ and $\lambda$.

- To simplify notation, we accumulate all weights in the matrix $\Theta = (\Theta^V, \Theta^\alpha, \Theta^\mu, \Theta^\lambda)$.

- We could think about the approach as just one large neural network with multiple outputs.
The HJB error:

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

The policy function error:

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

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 \( \alpha \).
Error criterion II

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

\[ \text{err}_{PF_1}(x; \Theta) \equiv \max\{0, G(x, \tilde{\alpha}(x; \Theta^\alpha))\} \]

\[ \text{err}_{PF_2}(x; \Theta) \equiv H(x, \tilde{\alpha}(x; \Theta^\alpha)) \]

the dual feasibility error:

\[ \text{err}_{DF}(x; \Theta) = \max\{0, -\bar{\mu}(x; \Theta^\mu)\} \]

and the complementary slackness error:

\[ \text{err}_{CS}(x; \Theta) = \bar{\mu}(x; \Theta)^T G(x, \tilde{\alpha}(x; \Theta^\alpha)) \]

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

\[ \mathcal{E}(x; \Theta) \equiv \left\| \text{err}_{HJB}(x; \Theta) \right\|_2^2 + \left\| \text{err}_{\alpha}(x; \Theta) \right\|_2^2 + \left\| \text{err}_{PF_1}(x; \Theta) \right\|_2^2 + \]

\[ + \left\| \text{err}_{PF_2}(x; \Theta) \right\|_2^2 + \left\| \text{err}_{DF}(x; \Theta) \right\|_2^2 + \left\| \text{err}_{CS}(x; \Theta) \right\|_2^2 \]
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 $\eta_k$ being the learning rate in the $k$-th epoch.
An Example
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:

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

Notice that \(c = (U')^{-1}(V'(k))\). With CRRA utility, this simplifies further to \(c = (V'(k))^{-\frac{1}{\gamma}}\).

We set \(\gamma = 2, \rho = 0.04, F(k) = 0.5 \times k^{0.36}, \delta = 0.05\).
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”:

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

• Details:

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

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

- 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

\[
\begin{array}{cccc}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
\end{array}
\]

Kernel

\[
\begin{array}{cc}
  w & x \\
  y & z \\
\end{array}
\]

Output

\[
\begin{array}{cc}
  aw + bx + ey + fz & bw + cx + fy + gz \\
  ew + fx + iy + jz & fw + gx + jy + kz \\
  cw + dx + gy + hz & gw + hx + ky + lz \\
\end{array}
\]

Figure 9.1: An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called “valid” convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.
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 unknown 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?
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).
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*.
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. \( \epsilon \)-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).
select randomly from among all the actions with equal probability, independently of the action-value estimates. We call methods using this near-greedy action selection rule \( \epsilon \)-greedy methods. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the \( Q_t(a) \) converge to \( q(\star)(a) \). This of course implies that the probability of selecting the optimal action converges to greater than 1, that is, to near certainty. These are just asymptotic guarantees, however, and say little about the practical effectiveness of the methods.

Exercise 2.1: In \( \epsilon \)-greedy action selection, for the case of two actions and \( \epsilon = 0.5 \), what is the probability that the greedy action is selected?

The 10-armed Testbed

To roughly assess the relative effectiveness of the greedy and \( \epsilon \)-greedy action-value methods, we compared them numerically on a suite of test problems. This was a set of 2000 randomly generated \( k \)-armed bandit problems with \( k = 10 \). For each bandit problem, such as the one shown in Figure 2.1, the action values, \( q(\star)(a), a = 1, \ldots, 10 \), were selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean \( q(\star)(a) \) unit variance normal distribution, as suggested by these gray distributions.

Figure 2.1: An example bandit problem from the 10-armed testbed. The true value \( q(\star)(a) \) of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean \( q(\star)(a) \) unit variance normal distribution, as suggested by these gray distributions.
2.3. The 10-armed Testbed were selected according to a normal (Gaussian) distribution with mean 0 and variance 1. Then, when a learning method applied to that problem selected action $A_t$ at time step $t$, the actual reward, $R_t$, was selected from a normal distribution with mean $q^\ast(A_t)$ and variance 1. These distributions are shown in gray in Figure 2.1. We call this suite of test tasks the 10-armed testbed. For any learning method, we can measure its performance and behavior as it improves with experience over 1000 time steps when applied to one of the bandit problems. This makes up one run. Repeating this for 2000 independent runs, each with a different bandit problem, we obtained measures of the learning algorithm's average behavior.

Figure 2.2 compares a greedy method with two "$\varepsilon$-greedy methods ($\varepsilon=0.01$ and $\varepsilon=0.1$), as described above, on the 10-armed testbed. All the methods formed their action-value estimates using the sample-average technique. The upper graph shows the increase in expected reward with experience. The greedy method improved slightly faster than the other methods at the very beginning, but then leveled off at a lower level. It achieved a reward-per-step of only about 1, compared with the best possible of about 1.55 on this testbed. The greedy method performed significantly worse in the long run because it (greedy)
A more general update rule

- Let’s think about a modified update rule:

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

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:

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

where

$$H_{n+1}(A_n) = H_n(A_n) + \alpha (1 - \pi_n(A_n)) (R_n(a) - \bar{R}_n)$$

$$H_{n+1}(a) = H_n(a) - \alpha \pi_n(a) (R_n(a) - \bar{R}_n) \text{ for all } a \neq A_n$$

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

All the methods we have discussed so far are dependent to some extent on the initial action-value estimates, $Q_1(a)$. In the language of statistics, these methods are biased by their initial estimates. For the sample-average methods, the bias disappears once all actions have been selected at least once, but for methods with constant $\varepsilon$, the bias is permanent, though decreasing over time as given by (2.6). In practice, this kind of bias is usually not a problem and can sometimes be very helpful. The downside is that the initial estimates become, in effect, a set of parameters that must be picked by the user, if only to set them all to zero. The upside is that they provide an easy way to supply some prior knowledge about what level of rewards can be expected.

Initial action values can also be used as a simple way to encourage exploration. Suppose that instead of setting the initial action values to zero, as we did in the 10-armed testbed, we set them all to +5. Recall that the $q_\ast(a)$ in this problem are selected from a normal distribution with mean 0 and variance 1. An initial estimate of +5 is thus wildly optimistic. But this optimism encourages action-value methods to explore. Whichever actions are initially selected, the reward is less than the starting estimates; the learner switches to other actions, being "disappointed" with the rewards it is receiving. The result is that all actions are tried several times before the value estimates converge. The system does a fair amount of exploration even if greedy actions are selected all the time.

Figure 2.3 shows the performance on the 10-armed bandit testbed of a greedy method using $Q_1(a) = +5$, for all $a$. For comparison, also shown is an $\varepsilon$-greedy method with $Q_1(a) = 0$. Initially, the optimistic method performs worse because it explores more, but eventually it performs better because its exploration decreases with time. We call this technique for encouraging exploration optimistic initial values. We regard it as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration. For example, it is not well suited to nonstationary problems because its drive for exploration is inherently zero.

![Figure 2.3: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\varepsilon = 0.1$.](image)
where \( \ln t \) denotes the natural logarithm of \( t \) (the number that \( e^{\pi/2} \approx 2.71828 \) would have to be raised to in order to equal \( t \)), \( N_t(a) \) denotes the number of times that action \( a \) has been selected prior to time \( t \) (the denominator in (2.1)), and the number \( c > 0 \) controls the degree of exploration. If \( N_t(a) = 0 \), then \( a \) is considered to be a maximizing action.

The idea of this upper confidence bound (UCB) action selection is that the square-root term is a measure of the uncertainty or variance in the estimate of \( a \)'s value. The quantity being max'ed over is thus a sort of upper bound on the possible true value of action \( a \), with \( c \) determining the confidence level. Each time \( a \) is selected the uncertainty is presumably reduced: \( N_t(a) \) increments, and, as it appears in the denominator, the uncertainty term decreases. On the other hand, each time an action other than \( a \) is selected, \( t \) increases but \( N_t(a) \) does not; because \( t \) appears in the numerator, the uncertainty estimate increases.

The use of the natural logarithm means that the increases get smaller over time, but are unbounded; all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time.

Results with UCB on the 10-armed testbed are shown in Figure 2.4. UCB often performs well, as shown here, but is more difficult than \( \varepsilon \)-greedy to extend beyond bandits to the more general reinforcement learning settings considered in the rest of this book. One difficulty is in dealing with nonstationary problems; methods more complex than those presented in Section 2.5 would be needed. Another difficulty is dealing with large state spaces, particularly when using function approximation as developed in Part II of this book. In these more advanced settings the idea of UCB action selection is usually not practical.

Exercise 2.8: UCB Spikes
In Figure 2.4 the UCB algorithm shows a distinct spike in performance on the 11th step. Why is this? Note that for your answer to be fully satisfactory it must explain both why the reward increases on the 11th step and why it decreases on the subsequent steps. Hint: if \( c = 1 \), then the spike is less prominent.\(^\ast\)
2.10 Summary

We have presented in this chapter several simple ways of balancing exploration and exploitation. The \( \varepsilon \)-greedy methods choose randomly a small fraction of the time, whereas UCB methods choose deterministically but achieve exploration by subtly favoring at each step the actions that have so far received fewer samples. Gradient bandit algorithms estimate not action values, but action preferences, and favor the more preferred actions in a graded, probabilistic manner using a soft-max distribution. The simple expedient of initializing estimates optimistically causes even greedy methods to explore significantly.

It is natural to ask which of these methods is best. Although this is a difficult question to answer in general, we can certainly run them all on the 10-armed testbed that we have used throughout this chapter and compare their performances. A complication is that they all have a parameter; to get a meaningful comparison we have to consider their performance as a function of their parameter. Our graphs so far have shown the course of learning over time for each algorithm and parameter setting, to produce a learning curve for that algorithm and parameter setting. If we plotted learning curves for all algorithms and all parameter settings, then the graph would be too complex and crowded to make clear comparisons. Instead we summarize a complete learning curve by its average value over the 1000 steps; this value is proportional to the area under the learning curve. Figure 2.6 shows this measure for the various bandit algorithms from this chapter, each as a function of its own parameter shown on a single scale on the x-axis. This kind of graph is called a parameter study. Note that the parameter values are varied by factors of two and presented on a log scale. Note also the characteristic inverted-U shapes of each algorithm's performance; all the algorithms perform best at an intermediate value of their parameter, neither too large nor too small. In assessing

Average reward over first 1000 steps

\[ \varepsilon \text{-greedy} \]

\[ \text{UCB} \]

\[ \text{gradient bandit} \]

\[ \alpha = 0.1 \]

Figure 2.6: A parameter study of the various bandit algorithms presented in this chapter. Each point is the average reward obtained over 1000 steps with a particular algorithm at a particular setting of its parameter.
Other algorithms

- Monte Carlo prediction.

- Temporal-difference (TD) learning:
  \[ V^{n+1}(s_t) = V^n(s_t) + \alpha (r_{t+1} + \beta V^n(s_{t+1}) - V^n(s_t)) \]

- SARSA ⇒ On-policy TD control:
  \[ Q^{n+1}(a_t, s_t) = Q^n(a_t, s_t) + \alpha (r_{t+1} + \beta Q^n(a_{t+1}, s_{t+1}) - Q^n(a_t, s_t)) \]

- Q-learning ⇒ 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.